VMS for Alpha Platforms Internals and Data Structures

Preliminary Edition
Volume 1

Ruth E. Goldenberg
and Saro Saravanan

VMS for Alpha Platforms

Internals and Data Structures

PRELIMINARY EDITION

VMS for Alpha Platforms Internals and Data Structures

PRELIMINARY EDITION

Volume 1

Ruth E. Goldenberg and Saro Saravanan

digital
DIGITAL PRESS

Copyright © 1992 by Digital Equipment Corporation.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, photocopying, recording, or otherwise, without prior written permission of the publisher.

Order number EY-L466E-P1.
ISBN 1-55558-095-5

The publisher believes the information in this preliminary publication was accurate at the time of writing; some information may have changed in the interval between writing and publication, and it remains subject to change without notice. Digital is not responsible for any inadvertent errors.

Printed in the United States of America.

The publisher offers discounts on quantity orders of this book. For information, please write:

Special Sales Department
Digital Press
One Burlington Woods Drive
Burlington, MA 01803

Cover design: Outside Designs
Copy editor: Alice Cheyer
Proofreader: MaryEllen Oliver
Compositors: Sarah Lemaire using VAX DOCUMENT;
Chiron, Inc. using PostScript
Production: Editorial Inc.
Printing: Hamilton Printing Company

A quotation from the following work appears as an epigraph in this book: excerpt from "The Hollow Men," in *Collected Poems 1909–1962* by T. S. Eliot, copyright 1936 by Harcourt Brace Jovanovich, Inc., copyright © 1964, 1963 by T. S. Eliot, reprinted by permission of Harcourt Brace Jovanovich and Faber and Faber Ltd.

C-bus is a trademark of Cromenco, Inc. Futurebus+ is an IEEE Standard. PostScript is a trademark of Adobe Systems Incorporated. CI, DDIF, DECnet, Digital, the Digital logo, DSSI, MSCP, TURBOchannel, VAX, VAX DOCUMENT, VAX MACRO, VAX RMS, VAX/VMS, VAXcluster, VMS, and XMI are trademarks of Digital Equipment Corporation.

Contents

This Preliminary Edition consists of several volumes. The present volume, the first to be published, contains the chapters that could be printed at this time. Subsequent volumes will be published later as more chapters are written. The number in brackets following the chapter title indicates that the chapter has been published and in which volume of the Preliminary Edition it appears.

Preface

Part I/INTRODUCTION

System Overview

I/O Architecture [Volume 1]
- The Generic I/O System
- Physical Address Space
- Tightly Coupled and Loosely Coupled I/O Interconnects

Part II/CONTROL MECHANISMS

Interrupts and Exceptions [Volume 1]
- Overview
- System Control Block
- Interrupts
- Exceptions
- Comparison of Exceptions and Interrupts
- The REI PALcode Routine

Hardware Interrupts and Machine Checks [Volume 1]
- Overview
- Powerfail Interrupt
- Machine Checks and Corrected Error Interrupts
- Processor-Specific Interrupts
- Interval Timer Interrupt
- I/O Interrupts
- Unexpected Interrupts and Exceptions

Software Interrupts [Volume 1]
- AST Delivery Interrupts
- Overview
- Software Interrupt Service Routines
- Fork Processing
- Kernel Processes
- Revision Information

Condition Handling

System Service Dispatching [Volume 1]
Overview
Transfer Routines for System Services in Executive Images
System Services That Do Not Change Mode
Change Mode Instructions
Change Mode Dispatching
Synchronous System and RMS Services
System Services in Privileged Shareable Images
Related System Services
Revision Information

ASTs [Volume 1]
Characteristics of ASTs
Architectural Support for ASTs
AST Data Structures
Enabling and Disabling AST Delivery
Creating an AST
Queuing an AST to a Process
Delivering an AST
Special Kernel Mode ASTs
System Use of Normal ASTs
Attention and Out-of-Band ASTs
Revision Information

Part III/SYNCHRONIZATION

Synchronization Techniques [Volume 1]
Overview
Synchronization at the Hardware Level
Synchronization at the Software Level
Elevated IPL
Spinlocks
Serialized Access
Mutual Exclusion Semaphores (Mutexes)
Further Information
Revision Information

Event Flags

Lock Management

Part IV/SCHEDULING AND TIME SUPPORT

Time Support

Scheduling

Process Control and Communication

Part V/MEMORY MANAGEMENT

Memory Management Overview

Memory Management Data Structures

Memory Management System Services

Paging Dynamics

Working Set List Dynamics

The Swapper

Pool Management

Part VI/INPUT/OUTPUT

Overview of the I/O Subsystem [Volume 1]
- Overview
- I/O Database
- Device Drivers
- Ancillary Control Processes
- I/O Support Routines

I/O System Services [Volume 1]
- Overview
- Device Drivers and Spinlocks
- Device Attributes
- Allocating and Deallocating Devices
- Assigning and Deassigning Channels
- $QIO System Service
- I/O Postprocessing
- Segmented Virtual and Logical I/O
- $CANCEL System Service
- Revision Information

I/O Processing [Volume 1]
- Device Driver Models in VMS
- The Start I/O Routine
- I/O Interrupt Dispatching
- Driver Kernel Processes
- Revision Information

Mailboxes

Miscellaneous I/O Topics

Part VII/LIFE OF A PROCESS

Process Creation

Image Activation and Exit

Translated Image Environment

Process Dynamics

Process Deletion

Part VIII/LIFE OF THE SYSTEM

The Modular Executive [Volume 1]
Overview
The Base Image SYS.EXE
The Base Image SYS$PUBLIC_VECTORS.EXE
Executive Images
Executive Image Data Structures
Executive Image Loading
Initialization of an Executive Image
Version Numbers
Dynamic Allocation and Deallocation of SPTEs
Revision Information

Bootstrap Processing [Volume 1]
Overview of System Initialization
Console Subsystem
Primary Bootstrap Program (APB)
Secondary Bootstrap Program (SYSBOOT)
Revision Information

Operating System Initialization and Shutdown

Error Handling

Power Failure and Recovery

Part IX/MISCELLANEOUS TOPICS

Logical Names

Miscellaneous System Services

APPENDIXES

System Processes and Privileged Images

Use of Listings and Map Files

SYSGEN Parameters

Naming Conventions [Volume 1]
- Public Symbol Patterns
- Object Data Types
- Facility Prefix Table
- Further Information

Data Structure Definitions

Size of System and P1 Virtual Address Spaces

Lock and Resource Use by VMS Components

Preface

Digital Equipment Corporation has announced a new 64-bit RISC architecture and VMS support for it. At the time of this writing, the architecture is known by its internal code name, Alpha, and does not yet have an official trademarked name. As a stand-in for an official name, this publication uses the adjective *censored.*

The *censored* architecture is a load/store architecture, with all operations done between registers. There are 32 integer registers and 32 floating-point registers, each 64 bits wide. A load or store instruction moves a longword or quadword between a register and memory. A memory operand is specified as a 16-bit signed displacement from the contents of an integer register.

Complex functions, such as interrupt and exception initiation, implemented in microcode on a VAX processor, are implemented in privileged architecture library (PALcode) routines. Written in *censored* instructions, these routines run in kernel mode with interrupts disabled. They extend the CPU architecture to provide a more complete environment for the operating system than the CPU alone could.

The subject of this publication is the VMS operating system that supports this RISC architecture. *Censored* VMS is ported from VAX/VMS Version 5.4-2. A majority of its source modules are based upon VAX/VMS V5.4-2 source modules. A typical VAX/VMS source module is slightly modified, compiled by a compiler that produces *censored* instructions, and linked to produce a *censored* image. Because much of the VMS executive is closely connected to specific architectural and hardware features, many executive modules have been more than slightly changed. Modules that interface directly with architectural features, for example, the scheduling code that swaps process contexts and the service routine for asynchronous system trap (AST) interrupts, have been entirely rewritten.

This is the first volume of the Preliminary Edition of *VMS for Alpha Platforms Internals and Data Structures*. It will be followed by additional volumes. Just as *censored* VMS has been ported from VAX/VMS Version 5.4-2, this publication has been ported from *VAX/VMS Internals and Data Structures: Version 5.2* (Digital Press, 1991).

The writing of this Preliminary Edition has been concurrent with the development of *censored* VMS. The chapters that make up this and other volumes do not all describe the same base level of *censored* VMS. Instead, each chapter documents a snapshot in time of the source modules relevant to that chapter. The revision information table at the end of some chapters lists the source modules documented in the chapter and indicates their approximate dates.

The chapters in the Preliminary Edition are numbered based on their position in the planned final book (see full Contents at the front of this volume). Thus, the chapters selected for updating here do not necessarily follow a continuous numerical sequence.

No data structure or subroutine described in these chapters is guaranteed to remain the same as the version identified in the revision information table at the end of the chapters. In fact, many data structures have changed since the chapters describing them were written. In particular, many byte and word fields have been promoted to longwords. It would be unwise to write code based on any particulars given here. Nonetheless, the authors believe that this Preliminary Edition will provide a conceptual description of *censored* VMS.

As more of the chapters are updated, additional preliminary volumes will be published. In the final book, all chapters will be updated to describe one particular released version of *censored* VMS.

The final *censored* VMS book will include some new chapters not in the VAX/VMS Version 5.2 book. As a result, the chapter numbers in the two books will not exactly correspond. In the preliminary volumes, therefore, references from one chapter to another are by title rather than number, for example, Chapter *Synchronization Techniques*. References from one chapter to tables and figures in another have been omitted in the preliminary volumes but will reappear in the final book.

The preliminary volumes have change bars (vertical bars in the inside margin) to highlight new material or material with substantive changes. Pages are perforated and punched so that they can be torn out and put into a notebook with the chapters of other preliminary volumes.

Organization of This Publication

This edition retains the organization of *VAX/VMS Internals and Data Structures: Version 5.2*. It is divided into nine parts, each of which describes a different aspect of the operating system.

- Part 1 presents an overview of the operating system and reviews those concepts that are basic to its workings.
- Part 2 describes the mechanisms used to pass control between user programs and the operating system, and within the system itself.
- Part 3 describes the synchronization methods of VMS.
- Part 4 describes scheduling, time support, and process control.
- Part 5 discusses memory management, with emphasis on system data structures and their manipulation by paging and swapping routines. It also describes management of dynamic memory, such as nonpaged pool.
- Part 6 contains an overview of the I/O subsystem, paying particular attention to the I/O-related system services.
- Part 7 describes the life cycle of a process: its creation, the activation and termination of images within its context, and its deletion.
- Part 8 describes the life of the system: its organization, initialization, error handling, and shutdown.
- Part 9 discusses the implementation of logical names and the internals of several miscellaneous system services.

- The appendixes include a summary of VMS data structures, layouts of system and P1 virtual address spaces, information on the use of listings and map files, the conventions used in naming symbols, and information about lock and resource use by various VMS components.

Conventions

A number of conventions are used throughout the text and figures of this book.

As previously described, the term *censored* identifies the architecture whose internal code name is Alpha. The version of VMS that supports it is identified here as *censored* VMS. Various hardware platforms are planned; they are identified here as SYSTEM_A, SYSTEM_B, and SYSTEM_C.

The term *executive* refers to those parts of the operating system that are loaded into and that execute from system space. The executive includes the system base images, SYS.EXE and SYS$PUBLIC_VECTORS.EXE, and a number of other loadable executive images. Unlike VAX/VMS executive images, all images loaded into *censored* VMS system space have the form of a loadable executive image. Because there is no longer a need to distinguish different types of executive image, this book generally shortens the term *loadable executive image* to *executive image*.

The terms *system* and *VMS system* describe the entire VMS software package, including privileged processes, utilities, and other support software as well as the executive itself. VMS consists of many different components, each a different file. One component is the system base image, SYS.EXE. Others are executive images, device drivers, command language interpreters, and utility programs.

The source modules from which these components are built and their listings are divided into facilities. Each facility is a directory on a source or listing medium containing sources and command procedures to build one or more components. The facility [DRIVER], for example, contains sources for most of the device drivers. The facility [SYSBOOT] contains sources for the secondary bootstrap program, SYSBOOT. The facility [SYS] contains the sources that make up the base images and many executive images.

This book identifies a [SYS] facility source module only by its file name. It identifies a module from any other facility by facility directory name and file name. For example, [SYSGEN]SYSGEN refers to the source for the system generation utility (SYSGEN). Appendix *Use of Listings and Map Files* discusses how to locate a module in the VMS source listings.

In general, the component called INIT refers to a module of that name in the executive and not to the volume initialization utility. When the latter is referenced, it is clearly specified.

This book identifies a macro from SYS$LIBRARY:LIB.MLB by only its name, for instance, WFIKPCH. The macro library of all other macros is specified.

The unmodified terms *process control block* and *PCB* refer to the software data structure used by the scheduler. The data structure that contains a process's hardware context, the hardware privileged context block (HWPCB), is always called the HWPCB.

The term *inner access modes* means those access modes with more privilege. The term *outer access modes* means those with less privilege. Thus, the innermost access mode is kernel and the outermost mode is user.

SYSGEN parameters include both the dynamic parameters, which can be changed on the running system, and the static parameters, whose changes do not take effect until the next system boot. These parameters are referred to by their parameter names rather than by the global locations where their values are stored. Appendix *SYSGEN Parameters* relates parameter names to their corresponding global locations.

Three conventions are observed for lists:

- In lists like this one, where no order or hierarchy exists, list elements are indicated by leading bullets (•). Sublists without hierarchy are indicated by dashes (–).
- Lists that indicate an ordered set of operations are numbered. Sublists that indicate an ordered set of operations are lettered.
- Numbered lists with the numbers enclosed in circles indicate a correspondence between the list elements and numbered items in a figure or example.

Several conventions are observed for figures. In all diagrams of memory, the lowest virtual address appears at the top of the page and addresses increase toward the bottom of the page. Thus, the direction of stack growth is depicted upward from the bottom of the page. In diagrams that display more detail, such as bytes within longwords, addresses increase from right to left. That is, the lowest addressed byte (or bit) in a longword is on the right-hand side of a figure and the most significant byte (or bit) is on the left-hand side.

Each field in a data structure layout is represented by a rectangle. In many figures, the rectangle contains the last part of the name of the field, excluding the structure name, data type designator, and leading underscore. A rectangle the full width of the diagram generally represents a longword regardless of its depth. A field smaller than a longword is represented in proportion to its size; for example, bytes and words are quarter- and half-width rectangles. A quadword is generally represented by a full-width rectangle with a short horizontal line segment midway down each side. In some figures, a rectangle the full width of the diagram represents a quadword. In these figures, bit position numbers above the top rectangle show numbers from 0 to 63 to indicate that the rectangle represents a quadword.

In almost all data structures, the data structure's full-width rectangles represent longwords aligned on longword boundaries. In a few data structures, a horizontal row of boxes represents fields whose sizes do not total a longword. Without this practice, most of the fields in this kind of structure would be split into two part-width rectangles in adjoining rows, because they are unaligned longwords.

A data structure field containing the address of another data structure in the same figure is represented by a bullet connected to an arrow pointing to the other structure. Where possible, the arrow points to the rightmost end of the field, that is, to bit 0. A field containing a value used as an index into that or another data structure is represented by an x connected to an arrow pointing to the indexed location.

Two conventions indicate elisions in a data structure layout. A specific amount of space is shown as a rectangle whose sides contain dots. Text within the rectangle indicates the amount of space it represents.

An indeterminate amount of space, often unnamed, representing omitted and undescribed fields, is indicated by a rectangle whose sides are intersected by short parallel horizontal lines.

Acknowledgments

Many people have contributed to this volume. Complete acknowledgments will appear in the final book.

Ruth E. Goldenberg
March 1992

Part I/Introduction

Chapter 2

I/O Architecture

Always design a thing by considering it in
its next larger context—a chair in a room,
a room in a house, a house in an environment,
an environment in a city plan.

Eero Saarinen

This chapter outlines a model of a generic [*censored*] I/O system and defines the basic terms used to describe its components. It also outlines other architectural aspects of I/O on a [*censored*] system.

2.1 The Generic [*Censored*] I/O System

This section presents a generic [*censored*] I/O system and introduces the terms that describe various system components. Only a functional description of system components is provided.

The functional building blocks of a [*censored*] system are hardware blocks and interconnects.

- A hardware block is an assembly of hardware components that implements a specific function.

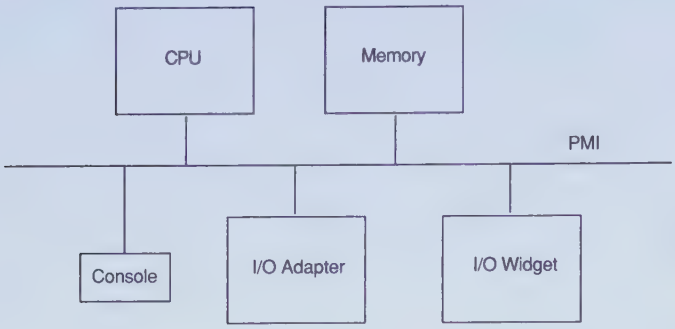
  A hardware block can be a part of a hardware module, a whole hardware module, or more than one hardware module. The term *hardware module* refers to a *physical* entity—it is a board with hardware components assembled on it.

- An interconnect is a physical medium that carries address, data, and control signals between various hardware blocks. A bus is one form of interconnect; a crossbar switch is another.

  A hardware block connected to an interconnect is called a node on that interconnect.

Figure 2–1 shows a generic [*censored*] system, and Figure 2–2 shows the [*censored*] I/O hardware blocks and interconnects. The major hardware blocks and interconnects shown are the following:

Figure 2–1 A Generic [*Censored*] System



- Central processing unit (CPU). This hardware block consists of a [*censored*] processor and cache memory along with cache management and addressing logic. Except where stated otherwise, the terms *processor* and *CPU* are used interchangeably in this book.

  The architecture provides for multiple CPUs; however, neither the hardware nor [*censored*] VMS currently supports multiprocessing.

- Memory. This hardware block contains random access memory (RAM) chips along with interface and control logic.

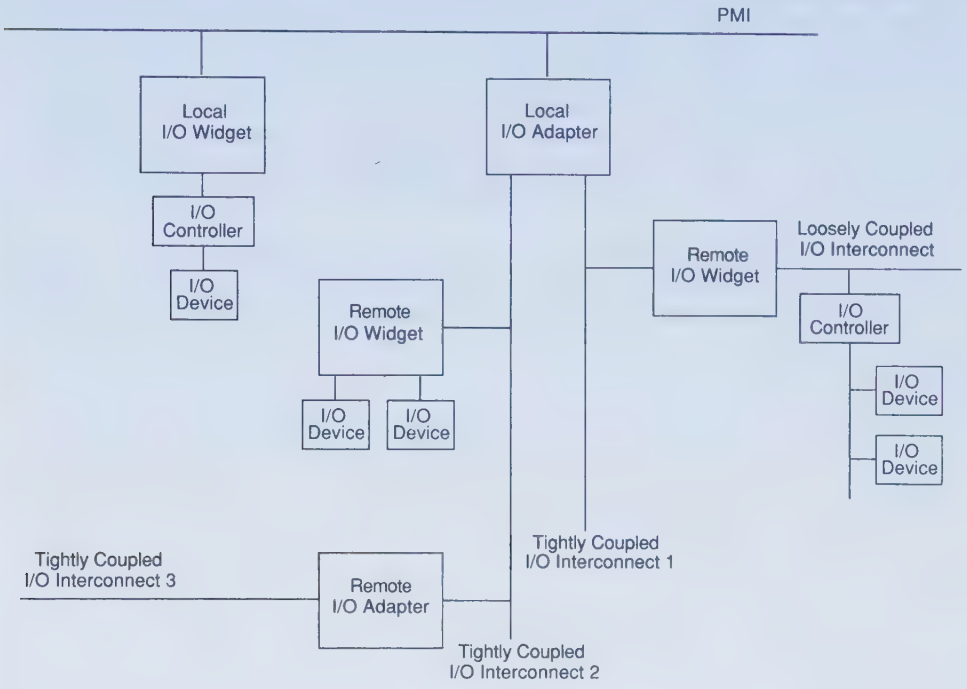
- Console. The console subsystem enables a human operator to monitor and control the system. It also supports system transitions such as power on, power failure, and bootstrap. Chapter *Bootstrap Processing* provides more information on consoles.

- Processor-memory interconnect (PMI). The CPU and memory communicate on this interconnect. I/O widgets and I/O adapters (defined later in this list) can also connect to the PMI.

- I/O interconnect. This is an interconnect whose primary purpose is I/O. There are two classes of I/O interconnect: tightly coupled I/O interconnects and loosely coupled I/O interconnects. Historically, the term *I/O bus* has referred to both classes of I/O interconnect. To avoid ambiguity, this book does not use the term.

  A tightly coupled I/O interconnect has an address space that is accessible to the CPU through direct or mailbox access, whereas a loosely coupled I/O interconnect does not. The terms *address space*, *direct access*, and *mailbox access* are defined in Section 2.2. Further discussion of the distinctions between tightly coupled and loosely coupled I/O interconnects is deferred to Section 2.3 because the distinctions are based on definitions that follow in this and subsequent sections.

Figure 2–2 *Censored* I/O Hardware Blocks and Interconnects



Tightly coupled I/O interconnects include the following:

— Extended memory interconnect (XMI)

— SYSTEM_A local I/O bus (L-bus)

— TURBOchannel

— Futurebus+

Loosely coupled I/O interconnects include the following:

— Small computer system interface (SCSI)

— Digital storage system interconnect (DSSI)

— Network interconnect (NI)

— Fiber distributed data interconnect (FDDI)

— Computer interconnect (CI)

— Storage interconnect (SI)

— Serial (EIA-232) lines

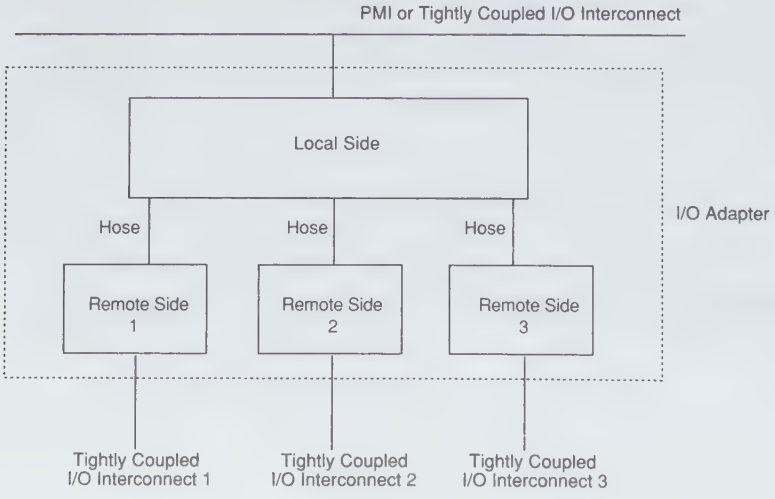
- I/O adapter. This hardware block connects one or more tightly coupled I/O interconnects to the PMI or to another tightly coupled I/O interconnect. Its primary purpose is to transform one interconnect protocol to another. An I/O adapter is also sometimes called a bridge.
- I/O widget. This hardware block connects one or more loosely coupled I/O interconnects to the PMI or to a tightly coupled I/O interconnect. The term *multichannel I/O widget* refers to an I/O widget to which multiple loosely coupled I/O interconnects are connected.

  The terminology that describes VAX systems does not differentiate tightly coupled from loosely coupled interconnects or adapters from widgets; *adapter* has referred to both types of hardware block, and *I/O bus* has referred to both types of interconnect. The terminology used here is a change from that tradition.

  The adjective *local* describes an I/O adapter or I/O widget that connects directly to the PMI. *Remote* describes an I/O adapter or I/O widget connected to a tightly coupled I/O interconnect. In descriptions of the components of an I/O adapter, *local* refers to a component that is physically closer to the PMI and *remote* to one farther away.

  Figure 2–3 shows the components of an I/O adapter. Minimally, an I/O adapter has a local side and a remote side connected by a hose. In this context, *side* refers to a hardware block that implements an interconnect protocol. The term *hose* refers to the data path between the local side and the remote side. An I/O adapter can have one or more local sides and one or more remote sides; each remote-local pair must be connected by a separate hose.
- I/O device (also called I/O device unit). This is the target of an I/O operation. It can be a physical device, such as a disk or tape, or a pseudo device. A pseudo device is one with no physical embodiment that is simulated by the operating system.

Figure 2–3 Components of an I/O Adapter



- I/O device controller. This hardware block provides control logic with interface registers to enable software to control the operation of one or more devices.

  An I/O device controller can be implemented as an integral part of a device or as part of an I/O widget that connects the device(s) to a tightly coupled I/O interconnect.

2.2 Physical Address Space

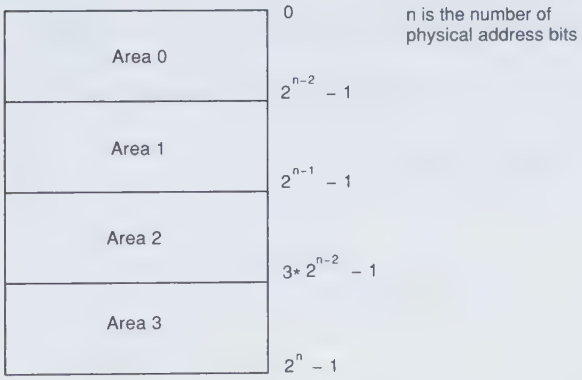
The collection of all physical locations addressable by a PMI node is its physical address space. The extent of this address space is defined by the number of physical address bits that the PMI node implements.

The [*censored*] architecture specifies that a [*censored*] processor can implement up to 48 physical address bits. Specific implementations can implement subsets of this range. A SYSTEM_A system CPU, for example, supports 34 address bits. Consequently, it can address byte locations 0 through $2^{34}-1$.

Architecturally, a processor's physical address space is divided into four areas of equal size, as shown in Figure 2–4. A processor can use an area either for memory references or for I/O location references, but not both. Area 0 must be used for memory references only. There is no such restriction on areas 1, 2, and 3; however, all areas used for memory references must be contiguous.

When a [*censored*] processor executes a load or store instruction, the source or target of the data can be either a memory space or an I/O space address.

Figure 2–4 Physical Address Space of a [*Censored*] Processor



2.2.1 Memory Space

The portion of a processor's physical address space through which it accesses memory is called its memory space.

A memory module is typically available in a standard size such as 32 MB, 64 MB, 128 MB, or 256 MB. Depending on system, configuration, and other factors, each memory module implements a unique portion of the memory space of the system.

[*Censored*] systems vary in the way address ranges are assigned to the memory modules. In one system, a hardware attribute such as its PMI slot number determines the address range that a memory module implements; in another system, console software assigns the address ranges during system initialization.

2.2.2 I/O Space

The portion of a processor's physical address space through which it accesses hardware interface registers is known as its I/O space.

A hardware interface register, or interface register for short, is the place where software interfaces with a hardware block. Every hardware block on a [*censored*] system, including CPU and memory, has a set of interface registers. The term *control/status register* (CSR) is sometimes used instead of the generic term *interface register*. In this book, *CSR* refers to a specific type of interface register that allows status determination and control of a hardware block.

A processor's I/O space is sometimes known as local I/O space to distinguish it from remote I/O space. The term *remote I/O space* describes the address space of any tightly coupled I/O interconnect on a [*censored*] system. The processor accesses remote I/O space through a hardware mailbox operation or by using mapped access, as described in Section 2.2.4.

A processor's I/O space is subdivided into smaller regions. A common subdivision, called node space, describes the part of physical address space through which a node's interface registers are accessed.

2.2.3 Device Driver I/O

A device driver typically interacts with an I/O device controller in the following manner to accomplish I/O:

1. The driver requests and initiates a device function.
2. The controller carries out the requested function.
3. The controller informs the driver of request completion.
4. The driver determines the status of the completed operation.

How each of these steps is actually accomplished depends on the design of the driver-controller interface. A simple interface consists of a set of interface registers that allows the driver to request and determine the status of I/O operations. In this model, driver-controller interaction is as follows:

1. The driver writes one or more interface registers to request and initiate a device function.
2. The controller carries out the requested function.
3. The controller generates an interrupt to inform the driver of request completion.
4. The driver reads one or more interface registers to determine the status of the completed operation.

A more sophisticated interface includes command and response buffers in addition to interface registers. The command and response buffers are allocated in memory and managed by the driver. In this model, driver-controller interaction is as follows:

1. The driver initializes a command buffer to request a device function; it can use multiple command buffers at one time. To initiate controller action, the driver writes to a controller interface register, notifying the controller that one or more command buffers are available.
2. The controller carries out the requested functions.
3. The controller records completion status in one or more response buffers and generates an interrupt to notify the driver of one or more request completions.
4. The driver determines the status of completed operations from the response buffers.

The latter model of I/O provides better performance for the following reasons:

- An interrupt disrupts the processor's instruction pipeline, thus affecting system performance. This model minimizes interrupts because one interrupt can signal the completion of multiple I/O operations.
- Interface register access is typically slower than access to memory or to the processor's cache. Read operations to interface registers usually take longer to complete than write operations (see Section 2.2.4). This model minimizes interface register access by having the controller write completion status to memory instead.

Chapter *Miscellaneous I/O Topics* provides more details on the driver-controller interface.

2.2.4 Interface Register Access

There are two types of interface register: registers that a processor accesses directly and those that it accesses only through a hardware mailbox operation.

2.2.4.1 Direct Access

An interface register with an address in local I/O space is directly accessible; it can be accessed with a load or store instruction. Software can directly access the interface register of a PMI node such as CPU, memory, local I/O adapter, or local I/O widget.

Software can directly access interface registers for remote I/O widgets only if the system implementation maps their addresses into the processor's I/O space. This special form of direct access is sometimes known as memory-mapped access, or simply, mapped access. The SYSTEM_B system, for example, uses mapped access.

2.2.4.2 Hardware Mailbox Access

Some systems do not support mapped access but instead support a mechanism called a hardware mailbox for accessing remote interface registers.

By this method, software creates a data structure in memory called a hardware mailbox and stores a command in it; this command, when issued over a remote interconnect, accesses a location on that interconnect. To initiate the access, software writes the hardware mailbox's physical address to a local I/O adapter's interface register. The local I/O adapter fetches the command from the mailbox and delivers it to the remote interconnect.

Hardware mailbox access overcomes the following disadvantages of mapped access:

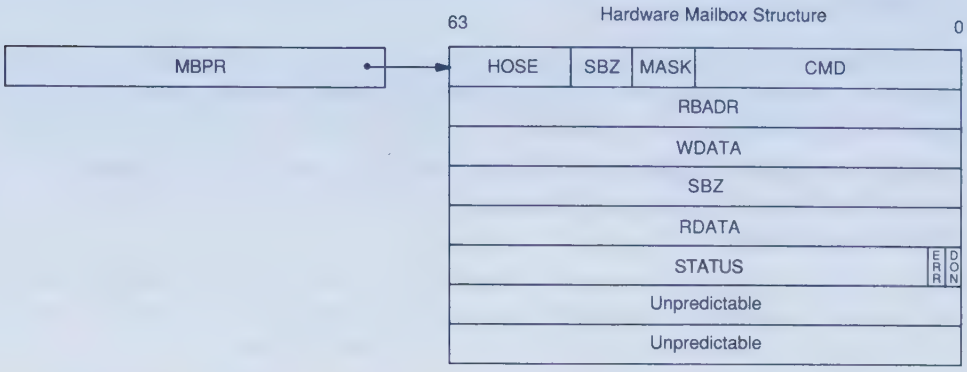
- A tightly coupled I/O interconnect (for example, Futurebus+) can have an address space that exceeds the maximum architecture-specified physical address space. All the address space of such an interconnect cannot be mapped into the processor's physical address space.
- The [*censored*] architecture supports only aligned quadword and longword accesses. Some I/O devices and interconnects require byte and word accesses.
- The PMI can be kept busy for the duration of a mapped access, thus delaying other PMI operations.

Hardware mailbox allocation and initialization can occur at any time, depending on how the structure is defined and used by the driver. A driver can use a unique hardware mailbox for each type of access it requires, or it can reuse the same mailbox for each access. The usage model and VMS support for mailbox access are discussed in Chapter *Miscellaneous I/O Topics*.

The [*censored*] architecture defines the layout of the hardware mailbox structure, which is shown in Figure 2–5. To access a remote interface register using this method, a driver initializes the hardware mailbox as follows:

1. It deposits a remote interconnect command in the CMD field that specifies an I/O operation such as a read or a write. This is the command that the local I/O adapter must deliver to the remote interconnect to which the target I/O widget is connected.
2. It writes a mask in the MASK field that specifies the size (byte, word, longword, or quadword) of the target location.
3. If the local I/O adapter connects to multiple tightly coupled I/O interconnects, the driver selects the appropriate tightly coupled I/O interconnect by entering a value in the HOSE field.
4. It writes the tightly coupled I/O interconnect address of the target interface register in the RBADR field.
5. If the remote interconnect command is a write, the driver initializes the WDATA field with data that is to be written to the target interface register.
6. All other fields of the hardware mailbox are zeroed.

Figure 2–5 Layout of a [*Censored*] Hardware Mailbox



After initializing the hardware mailbox, the driver initiates the mailbox operation by writing the physical address of the hardware mailbox to the mailbox pointer register (MBPR), an interface register on a local I/O adapter. This action triggers the local I/O adapter to read the hardware mailbox and perform the requested remote access.

After writing to the MBPR, the driver polls the DON bit in the STATUS field. For a write operation, a local I/O adapter can set the DON bit immediately after it has read the hardware mailbox or only after the remote write operation has actually completed, depending on I/O adapter implementation; the former scheme results in increased performance. For a read operation, however, the local I/O adapter can set the DON bit only after it has written the requested data to the RDATA field of the hardware mailbox.

For read operations only, the local I/O adapter can set the ERR bit to indicate an error. If ERR is set, additional error status is contained in the STATUS field.

2.2.5 Direct Memory Access

During an I/O operation, a controller can access data, command, and response buffers in memory without the intervention or assistance of the processor. Such an access is called direct memory access (DMA). If the controller is associated with a local I/O widget, DMA is a simple PMI operation.

A controller associated with a remote I/O widget requires the assistance of one or more intervening I/O adapters to access memory. Consider, for example, the remote I/O widget connected to tightly coupled I/O interconnect 1 in Figure 2–2. Tightly coupled I/O interconnect 1 connects to the local I/O adapter and there is no intervening I/O adapter. To assist in DMA, the local I/O adapter must transform the controller's request to access a location in the tightly coupled I/O interconnect's address space into a PMI request to access a memory location.

In other words, the local I/O adapter must map a portion of the tightly coupled I/O interconnect's address space into the processor's memory space.

2.3 Tightly Coupled and Loosely Coupled I/O Interconnects

The distinctions between tightly coupled and loosely coupled I/O interconnects are not obvious or precise. Refer to the lists in Section 2.1, where a number of I/O interconnects are classified. The following are general guidelines for classification:

- A tightly coupled I/O interconnect has an address space that is accessible to the CPU through direct or mailbox access, whereas a loosely coupled I/O interconnect does not.

  A loosely coupled I/O interconnect always has an associated I/O widget that connects it to a tightly coupled I/O interconnect or to the PMI. The CPU has direct or mailbox access to the I/O widget's interface registers.

- A tightly coupled I/O interconnect is typically limited in range to a system cabinet, whereas a loosely coupled I/O interconnect generally extends beyond the boundary of the system cabinet.

Part II/Control Mechanisms

Chapter 3

Interrupts and Exceptions

By indirections find directions out.

Shakespeare, *Hamlet,* 2, i

This chapter describes the *censored* architectural mechanisms for initiation of and return from an interrupt or exception. It summarizes VMS use of the mechanisms.

3.1 Overview

During system operation, events occur that require the execution of software other than the current thread of execution. The processor responds to such events by altering the control flow from the current thread of execution. Where control is transferred is determined by the system control block (SCB). The SCB contains an entry for each event specifying the address to which control is to be transferred.

Some of these events are unrelated to the current thread of execution and are asynchronous to it; these events are called interrupts. Other events, called exceptions, are triggered by the current thread and are typically synchronous to it.

Most hardware interrupts are requested by signals from devices external to the processor when they need attention from the operating system. The hardware interrupt capability makes it unnecessary for the processor to poll the device to determine whether its state has changed. A few types of hardware interrupt are requested by signals from components within the processor, such as the interval timer.

A software interrupt is an interrupt requested by kernel mode code rather than by an external device. The VMS executive is interrupt-driven and requests software interrupts to schedule operating system functions.

To enable arbitration among concurrent interrupt requests and their servicing, each interrupt request has an associated interrupt priority level (IPL). When an interrupt is granted, processor IPL is raised to that of the interrupt. When the processor IPL is at or above that of the interrupt request, the interrupt is blocked. A processor executing a privileged architecture library (PALcode) routine cannot be interrupted.

When an interrupt is requested, the processor determines whether its associated IPL permits it to interrupt:

- If the processor is running at an IPL equal to or higher than that of the interrupt request, the interrupt request is deferred until processor IPL drops below the IPL of the request.
- If the processor is running at a lower IPL than that of the interrupt request, the interrupt is granted.

To grant the interrupt, a typical [*censored*] processor may drain the instruction pipeline and complete any outstanding load instructions (these are CPU-specific steps). It then transfers control to a PALcode routine that saves processor state and dispatches to the interrupt's service routine through the SCB entry associated with the interrupt. Chapter *System Overview* briefly describes PALcode.

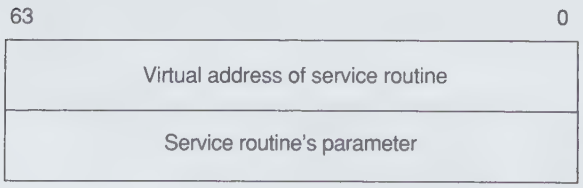
An exception is the processor's response to an anomaly or error it encounters while fetching or executing an instruction, for example, an access violation in a load or store instruction. An exception occurs in direct response to a particular instruction sequence and would occur again if the instruction were repeated under the same circumstances. (Note, however, that arithmetic exceptions are imprecise; there can be a delay between execution of the instruction that incurs the exception and initiation of the exception.) When an exception is initiated, the processor transfers control to a PALcode routine that saves processor state and dispatches to the exception's service routine through the SCB entry associated with the exception.

In general, each type of exception and interrupt has its own PALcode entry point. Most converge to common code, which this chapter calls the initiate-exception-or-interrupt (IEI) PALcode routine.

3.2 System Control Block

Operating system software initializes the SCB, and the [*censored*] IEI PALcode routine dispatches all interrupts and exceptions through it. Each exception and interrupt has a unique entry, identified by its offset from the beginning of the SCB. Figure 3–1 shows the organization of an SCB entry.

Figure 3–1 System Control Block Entry



Each entry consists of two quadwords. The first quadword, called the vector, contains the virtual address of a service routine for that exception or interrupt. Because no mapping context is switched at interrupt and exception initiation, the architecture requires that a particular service routine exist at the same virtual

address in every process context. VMS deals with this requirement by placing all interrupt and exception service routines in system space.

The second quadword, called the parameter, is passed in R3 as an argument to the service routine. The [*censored*] architecture does not define its contents. Except for parameters of I/O interrupt entries, each [*censored*] VMS parameter contains an address that can serve as a linkage section pointer for the service routine. (An interrupt or exception service routine written in anything other than [*censored*] assembly language begins with a compiler directive describing this linkage, which differs from that specified by the [*censored*] calling standard.)

Typically, [*censored*] VMS stores the address of the service routine's procedure descriptor, thereby creating a standard linkage pair in the SCB entry. For an I/O interrupt, [*censored*] VMS stores information that enables the I/O interrupt service routine to dispatch to a device-specific routine.

The SCB is page-aligned and occupies 8K bytes. The system control block base (SCBB) processor register contains the page frame number of the page the SCB occupies. The IEI PALcode routine calculates the address of a particular entry using the contents of the SCBB processor register and the offset into the SCB of the entry. This design enables executive software to place the SCB in memory known to be good at system initialization. If the SCB were required to be at a fixed location, and if that memory had uncorrectable errors, the system would be unable to run. During system initialization, VMS allocates a physical page of memory for the SCB and a system page table entry (SPTE) to map it, initializes SCBB and the SPTE, and stores its starting virtual address in global location EXE$GL_SCB.

Figure 3–2 shows the general organization of the SCB. It contains entries for faults and traps (which are types of exceptions), software interrupts, asynchronous system trap (AST) interrupts, processor hardware interrupts, and I/O interrupts. The macro $HWSCBDEF defines symbolic offsets for its entries.

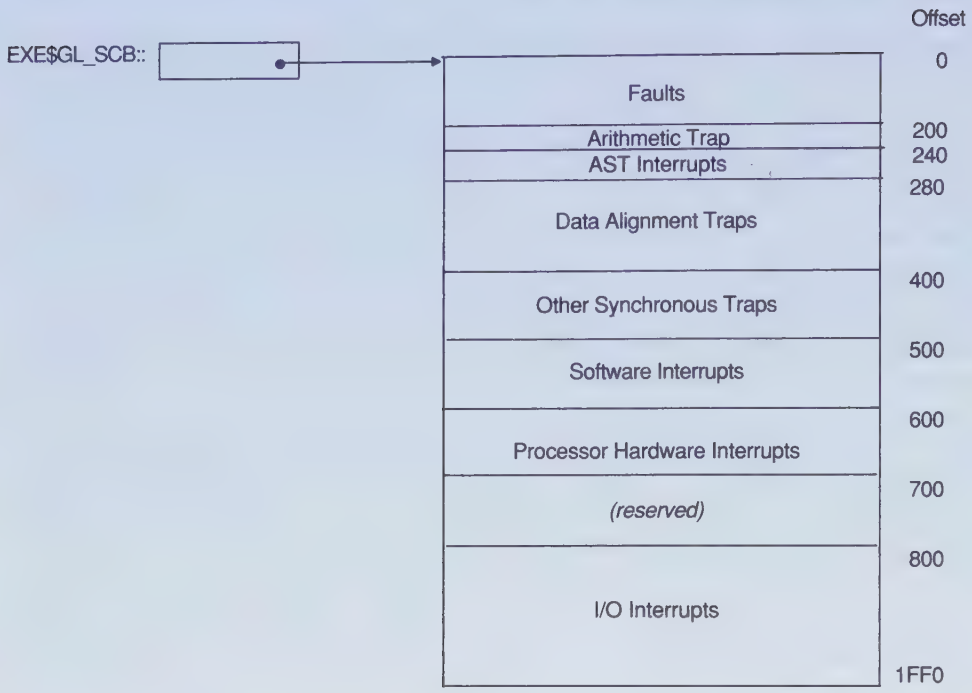
The executive handles most exceptions in a uniform way. Some exceptions, however, result in special action. Chapter *Condition Handling* lists the [*censored*] exception SCB entries, describes VMS handling of most exceptions, and summarizes responses to special exceptions.

Chapter *Software Interrupts* contains more details about software and AST interrupts and summarizes their service routines.

Chapter *Hardware Interrupts and Machine Checks* summarizes processor hardware interrupts and their service routines.

Chapters *Hardware Interrupts and Machine Checks* and *I/O Processing* describe the assignment of I/O interrupt SCB entries and I/O interrupt service routines.

Figure 3–2 System Control Block Organization



3.3 Interrupts

The [censored] architecture provides 16 hardware IPLs, from IPL 31 down to IPL 16. Interrupts at the higher levels are primarily for processor errors and power failure. The middle levels are primarily for interrupts from external adapters and I/O devices. The lower levels are unused.

There is no one-to-one correspondence between IPL and hardware interrupt vector. The SCB contains multiple entries whose interrupts are at the same hardware IPL.

There is, however, a one-to-one correspondence between IPL and software interrupt entry. The [censored] architecture provides 15 entries in the SCB for software interrupts at IPLs 1 through 15.

3.3.1 Interrupt Requests

The [censored] architecture provides a means for kernel mode code and system console commands to request software interrupts.

Kernel mode code requests a software interrupt at a particular IPL by invoking a PALcode routine to write that IPL into the software interrupt request register (SIRR). VAX MACRO code generally uses the SOFTINT macro, which expands into the following VAX MACRO statement:

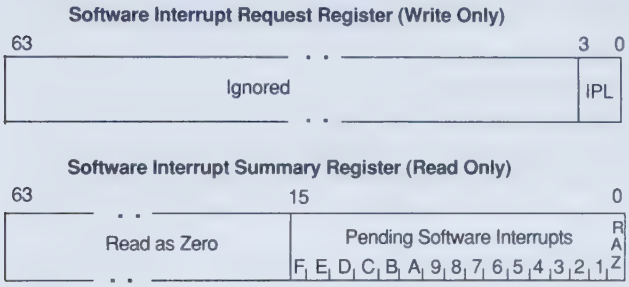
```
MTPR     ipl,S^#PR$_SIRR
```

When compiled, that statement generates instructions that load R16 with the IPL and perform a `CALL_PAL MTPR` whose destination is the SIRR processor register. The following CPU console command can also write the SIRR processor register:

```
D SIRR ipl
```

Writing to the SIRR processor register causes the bit with the same number as the IPL to be set in another processor register, the software interrupt summary register (SISR). Figure 3–3 shows the layouts of these two registers. At any given time, the SISR processor register contains a bit set for each level at which a software interrupt has been requested but not yet granted. The processor reads the SISR processor register to test for pending software interrupts. When the processor grants a software interrupt request, it clears the corresponding bit in the SISR processor register.

Figure 3–3 Layouts of Software Interrupt Request Register and Software Interrupt Summary Register



The [*censored*] architecture provides four entries in the SCB for AST interrupts, one for each access mode. An AST interrupt is requested by each of the following PALcode routines whenever it detects that a pending AST is deliverable: the REI, SWASTEN, MTPR_IPL, MTPR_ASTSR, and MTPR_ASTEN routines. Chapters *Software Interrupts* and *ASTs* provide more information.

To block interrupts of any type, kernel mode code can raise IPL to that of the highest interrupt to be blocked. The [*censored*] architectural concept of an interrupt includes the idea that an interrupt request is expected to persist until serviced or until the device withdraws the request.

3.3.2 Interrupt Dispatching

When an interrupt is requested and granted, the IEI PALcode routine takes the following steps:

1. The IEI PALcode routine switches to the kernel stack, if not already there. If the system was running on an outer access mode stack, the routine saves the contents of the stack pointer (SP) register in the stack pointer internal processor register for that mode. On a [*censored*] CPU type that does not implement these processor registers, the IEI PALcode routine instead stores SP in a save area in the process's hardware privileged context block (HWPCB).

It loads the SP from the HWPCB save area for the kernel stack. (There is no KSP processor register on any [*censored*] processor.)

2. It aligns the kernel stack to a 64-byte boundary.

3. It saves the access mode and IPL that were current prior to the interrupt by pushing onto the stack the processor status (PS) register. It records the previous alignment of the kernel stack in the quadword containing the saved PS. It also pushes onto the stack the program counter (PC) current at the time of the interrupt, that is, the address of the next instruction to be executed in the interrupted thread.

   Saving the PC and PS preserves state so that the interrupted thread of execution can continue after the interrupt is dismissed. Saving the alignment enables the SP to be restored to its previous value when the interrupt is dismissed.

   If the kernel stack is invalid when the IEI PALcode routine tries to push the interrupt frame onto the stack, the processor enters the kernel-stack-not-valid restart sequence. Chapter *Power Failure and Recovery* describes error restart processing.

4. The IEI PALcode routine saves R2 through R7 on the stack, freeing them to pass information to the interrupt service routine. The 64-byte set of information saved on the stack, shown in Figure 3–4, is called an interrupt or exception stack frame. The macro $INTSTKDEF defines symbolic offsets for its fields.

   As a result of the 64-byte alignment performed in step 2, the 64-byte interrupt or exception stack frame resides within one page, simplifying the IEI and REI PALcode routines.

5. On some [*censored*] implementations, the IEI PALcode routine executes a STQ_C instruction (writing the contents of a stack frame quadword back to itself) to clear the lock flag register in case it was set. As described in Chapter *Synchronization Techniques*, the [*censored*] architecture requires that the lock flag register be cleared at initiation of an interrupt or exception, either by hardware or PALcode. Clearing the flag enables a STx_C instruction within an interrupted LDx_L/STx_C instruction sequence to detect that the sequence has been interrupted and that the store would not be atomic and should not occur.

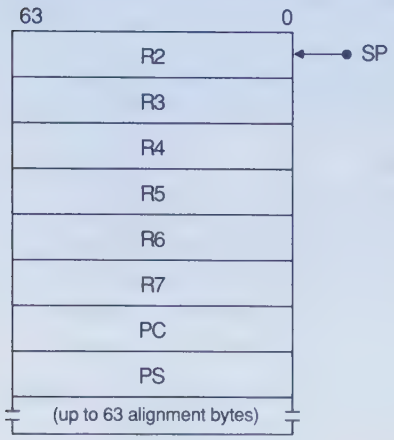
6. The IEI PALcode routine determines the offset into the SCB of the entry corresponding to the interrupt it is servicing.

7. It loads the address of the service routine from the first quadword of the SCB entry into R2 and loads R3 from the second quadword.

8. The IEI PALcode routine constructs a new PS whose IPL is that associated with the interrupt. Its current mode is kernel, the mode in which the interrupt will be serviced. Its software bits are zero.

9. When a software interrupt is dispatched, the IEI PALcode routine clears the bit in the SISR processor register corresponding to the IPL of the interrupt.

Figure 3–4 Layout of an Interrupt or Exception Stack Frame



10. The IEI PALcode routine transfers control to the interrupt service routine, whose address is in R2.

The interrupt service routine executes. An interrupt service routine written in [*censored*] assembly language uses R3 as a linkage section pointer. The prologue code for a VAX MACRO interrupt service routine assumes R3 contains a procedure descriptor address and copies it to another register for use as a linkage section pointer.

The interrupt service routine exits with the [*censored*] instruction `CALL_PAL REI`. (REI stands for return from exception or interrupt.) The REI PALcode routine, described in Section 3.6, restores the registers from the interrupt stack frame, restores the previous SP, and resumes the interrupted thread of execution (a process or lower priority interrupt service routine) at the restored PC and with the restored PS.

3.3.3 Restrictions Imposed on Interrupt Service Routines

An interrupt service routine runs at elevated IPL on the kernel stack of the current process. Which process is current at the time of an interrupt is indeterminate. Consequently, an interrupt service routine must be capable of executing in the environment of the most constrained possible process, the system process. Chapter *System Overview* describes this limited environment.

Several restrictions are imposed on interrupt service routines by either the [*censored*] architecture or VMS. Many of these result from the limitations of the system process and execution at elevated IPL. The following list indicates some of the constraints placed on an interrupt service routine. The description of the system process in Chapter *System Overview* contains a more general list of these and other restrictions. Chapter *Synchronization Techniques* describes synchronization rules applicable to an interrupt service routine.

- To reduce overhead, no context switch occurs with an interrupt. Therefore, VMS requires that instructions executed and data referenced by an interrupt service routine be in system space. An interrupt service routine may not refer to process address space: it could be executing in the system process context,

which has no per-process address space, or the target address could be in a page not currently part of the process's working set. Either case would cause a system crash.

- An interrupt service routine should be brief and do as little processing as possible at elevated IPL to minimize the length of time the servicing of other interrupts is blocked.
- An interrupt service routine must save any registers it uses other than R2 through R7, which are saved by the IEI PALcode routine; otherwise, the contents of registers in use by the interrupted thread of execution could be overwritten and thus corrupted. An interrupt service routine written in a language other than *censored* assembly language leaves this detail to the compiler.
- Prior to exiting, an interrupt service routine must remove anything it pushed onto the stack and restore any additional saved registers. Otherwise, the REI PALcode routine would be unable to restore correctly the interrupt stack frame registers and to resume the interrupted thread of execution. An interrupt service routine written in a language other than *censored* assembly language leaves this detail to the compiler.
- An interrupt service routine should be conservative in its use of stack space to minimize the chance of overflowing the kernel stack, which could be quite full at the time of the interrupt.
- VMS does not allow an interrupt service routine (or any other code) that executes above IPL 2 to access pageable routines or data structures. The page fault exception service routine generates a fatal bugcheck if a page fault occurs while IPL is above 2.
- VMS generates a fatal bugcheck if an interrupt service routine (or any other code) that executes above IPL 2 incurs an exception that would otherwise be dispatched to a condition handler.
- Although an interrupt service routine can raise IPL, it may not lower IPL below the level at which the original interrupt occurred. An interrupt service routine that lowers IPL below that of its interrupt enables itself to be reentered by another interrupt and breaks its own synchronization.

3.4 Exceptions

When an exception is detected, the IEI PALcode routine takes the following steps:

1. It determines on which stack the exception is to be serviced. Which stack depends on the access mode in which the exception occurred and the type of exception.

   An unaligned data exception is serviced on the stack current when the exception occurred. A change mode exception is generally serviced on the stack of its target mode. All other exceptions are serviced on the kernel stack.

Use of the kernel stack for most exceptions avoids the possibility of attempting to report an exception on, for example, the user stack, only to find that it is corrupted in some way (invalid or otherwise inaccessible), resulting in another exception. If the kernel stack is invalid when IEI PALcode tries to push the exception frame onto the new stack, the processor enters the kernel-stack-not-valid restart sequence (see Chapter *Power Failure and Recovery*).

2. If necessary, the IEI PALcode routine switches stacks, saving the previous SP in a processor register or the HWPCB, as described in Section 3.3.2. It aligns the stack to a 64-byte boundary.

3. It pushes the PC and PS onto the new stack. The exception PC that it pushes depends on whether the exception is a fault or trap:

   — For a fault, the IEI PALcode routine pushes the address of the faulting instruction onto the stack. When a fault is dismissed through the REI PALcode routine, the faulting instruction reexecutes. A fault is triggered, for example, by an attempt to access a location in a way prohibited to the current access mode (access violation), an attempt to access a location whose page table entry is not valid (translation-not-valid), or an attempt to execute a floating-point instruction when floating-point is disabled.

   — For a trap, the IEI PALcode routine pushes the address of the next instruction onto the new stack. An instruction that causes a trap does not reexecute when the exception is dismissed. Illegal instructions, unaligned data, and change mode to kernel are examples of traps. These are called synchronous traps to distinguish them from arithmetic traps.

   — An arithmetic trap occurs as the result of performing an arithmetic or conversion operation that incurs one of several possible errors. When an arithmetic exception condition is detected, several other instructions can be in various stages of execution. These instructions must complete executing before the arithmetic exception is initiated and can generate more arithmetic errors, possibly as the result of erroneous data forwarded from the instruction incurring the first trap. Consequently, arithmetic traps are imprecise, and a particular trap can represent multiple arithmetic exceptions. The IEI PALcode routine passes information in R5 summarizing the multiple arithmetic exceptions represented by the trap.

     The IEI PALcode pushes the address of the instruction that was the next to be executed when the trap was detected rather than the instruction following the one that incurred the exception.

4. The IEI PALcode routine saves R2 through R7 on the stack, forming the rest of the exception frame.

5. As described in Section 3.3.2, it may execute a STQ_C instruction to clear the lock flag register.

6. It loads the address of the service routine from the first quadword of the SCB entry into R2 and loads R3 from the second quadword.

7. The IEI PALcode routine constructs a new PS whose IPL is unchanged. Its current mode is set to the mode in which the exception will be serviced. Its software bits are zero.

8. Depending on the exception type, it loads exception-specific information into R4 and R5.
9. The IEI PALcode routine transfers control to the exception service routine, whose address is in R2.

The exception service routine executes. It eventually exits by executing a CALL_PAL REI instruction to dismiss the exception.

The REI PALcode routine, described in Section 3.6, restores the registers saved in the exception frame, including the PC and PS, restores the previous SP, and resumes the thread of execution that incurred the exception.

3.5 Comparison of Exceptions and Interrupts

The following list contrasts exceptions and interrupts.

- An interrupt occurs asynchronously to the currently executing instruction stream and is serviced between individual instructions. An exception occurs synchronously as a direct effect of the execution of an instruction. Except for imprecise arithmetic exceptions, an exception is serviced before the next instruction.
- Interrupts are generally systemwide events that cannot rely on support of a process in their service routines. Exceptions are generally a part of the currently executing process. Exception servicing is an extension of the instruction stream that is currently executing on behalf of that process.
- Interrupts cause an IPL change. Other than machine check exceptions, exceptions do not cause an IPL change.
- An interrupt is blocked while the processor executes at an IPL at or above the IPL associated with the interrupt. Exceptions cannot be blocked by elevated IPL.
- Certain exception service routines (for example, those for exceptions that can be passed to user-written condition handlers) record the previous access mode in bits <1:0> of the new PS. (Unlike the VAX processor status longword register, the [*censored*] PS does not include an architecturally defined previous mode field. The [*censored*] architecture does, however, leave the definition of the low three bits of the PS to software.) No interrupt service routine sets these bits. This difference between exceptions and interrupts reflects the fact that interrupts are not related to the interrupted instruction stream.

3.6 The REI PALcode Routine

Every interrupt or exception service routine exits by executing a CALL_PAL REI instruction, which causes dispatch to the REI PALcode routine. The [*censored*] architecture limits the types of transitions from one access mode to another; the CALL_PAL REI instruction is the only way to change access mode to a less privileged one. This property of CALL_PAL REI, in conjunction with the architectural constraint that an inner access mode will not be interrupted to deliver an AST to an outer mode, makes the REI PALcode routine a logical place to test whether an AST delivery interrupt should be requested.

Execution of a CALL_PAL REI instruction is a common way for IPL to be lowered. Because lowering IPL can make a pending interrupt deliverable, hardware and software interrupts are often delivered after a CALL_PAL REI instruction is executed.

Some protection and privilege checks are incorporated into the REI PALcode routine to prevent the system from entering illegal or inconsistent states. CALL_PAL REI is not a privileged instruction, and these checks prevent, for example, an attempt to enter a more privileged access mode.

The REI PALcode routine takes the following steps:

1. It checks that the stack is aligned to a 64-byte boundary, generating an illegal PALcode operand trap if it is not.
2. The PALcode routine checks that, if the mode is not kernel, the saved PS IPL and must-be-zero bits are zero. If the bits are not zero, the routine generates an illegal PALcode operand trap. This test prevents any mode other than kernel from raising IPL and also detects some forms of stack corruption.

   Note that, unlike VAX REI microcode, the REI PALcode routine does not prevent kernel mode code from executing an REI to a higher IPL; kernel mode code is assumed to be written correctly in this regard.
3. As described in Section 3.3.2, the routine may execute a STQ_C instruction to clear the lock flag register.
4. It checks that the saved PS current mode is no more privileged than the current PS current mode. If the attempted mode transition is illegal, the routine generates an illegal PALcode operand trap. This test prevents an attempt to REI to a more privileged mode.
5. It stores the interrupt/exception stack frame contents into temporary internal registers and restores SP to its value prior to the IEI alignment and push of the interrupt/exception stack frame.
6. The PALcode routine saves the contents of the updated SP register either in a processor register (ESP, SSP, or USP) or, for kernel mode or any mode on a processor without the stack pointer processor registers, in the HWPCB. This step records the pointer into the current access mode's stack.

   The REI PALcode routine loads SP from the appropriate *x*SP processor register. This step restores the pointer into the now-current stack. On a [*censored*] CPU type that does not implement these processor registers, the REI PALcode routine instead loads SP from the appropriate quadword in the process's HWPCB.
7. It restores R2 through R7, PC, and PS from the temporary locations in which it placed them in step 5.
8. It tests for a pending interrupt of high enough priority to interrupt the thread about to be resumed or an AST that could be delivered, given the IPL and access mode being restored. If any exists, it occurs before the next instruction.
9. Unless another interrupt occurs, execution resumes with the instruction being executed at the time of the interrupt or exception.

Chapter 4

Hardware Interrupts and Machine Checks

While I nodded, nearly napping, suddenly there came a tapping,
As of someone gently rapping, rapping at my chamber door.

Edgar Allen Poe, *The Raven*

The VMS operating system is often described as interrupt-driven and non-monolithic. Hardware interrupts notify VMS of such important events as device operation completion, hardware error and alert conditions, and power failure. In addition, the interval timer interrupt allows VMS to keep system time. Machine checks inform VMS of serious hardware error conditions.

This chapter provides an overview of hardware interrupts, hardware interrupt priority levels (IPLs), and machine checks.

4.1 Overview

As described in Chapter *Interrupts and Exceptions*, many hardware interrupts are requested by signals from devices external to the CPU module when they need attention from the operating system. Hardware interrupts are requested by controllers, widgets, and adapters. In addition, the processor itself can request some hardware interrupts.

The *censored* architecture provides 16 priority levels, 16 through 31, for hardware interrupts and 15 priority levels, 1 through 15, for software interrupts. Chapter *Software Interrupts* describes software IPLs.

When a hardware interrupt occurs, the interrupted processor raises its priority to the IPL associated with the interrupt. Table 4–1 provides a summary of hardware IPLs and their use by VMS. IPLs 16 through 19 are unused by VMS. Corrected errors are reported at IPL 20. Device interrupts occur at IPLs 20 through 23. Interval timer interrupts occur at IPL 22. Urgent conditions such as machine checks and power failure are reported at the highest priority levels, IPLs 24 through 31. A power failure is reported at IPL 30. IPL 31 blocks all interrupts when required.

Software running in kernel mode can raise and lower the priority of the processor by executing the `CALL_PAL MTPR_IPL` instruction, as described in Chapter *Interrupts and Exceptions*. Thus, software has the ability to block hardware interrupts as necessary. For example, a device driver raises IPL to 31 to block the powerfail interrupt and synchronize with powerfail recovery.

Table 4–1 Hardware Interrupt Priority Levels and Their Use

IPL	Name	Use
31	IPL$_POWER	Block all interrupts
	IPL$_EMB	Synchronize error logging
	IPL$_MCHECK	Synchronize machine check processing
	IPL$_MEGA	Synchronize miscellaneous structures
30	—	Powerfail interrupt
30–24	—	System- and processor-specific error interrupts
23–20	—	I/O interrupts
22	IPL$_HWCLK	Interval timer interrupt
20	—	Corrected error interrupts
19–16	—	Unused

A *censored* processor's response to any interrupt request, hardware or software, is similar. If the processor's priority and state permit the requested interrupt to be granted, the processor transfers control to a privileged architecture library (PALcode) routine. This routine saves the processor state on the current process's kernel stack and dispatches through the system control block (SCB) entry associated with the interrupt to its service routine.

Table 4–2 summarizes hardware interrupts and machine checks and their service routines. The following sections provide brief descriptions.

4.2 Powerfail Interrupt

CPU hardware requests a powerfail interrupt at IPL 30 when there is a drop in operating voltage. EXE$POWERFAIL, in module POWERFAIL, is the powerfail interrupt service routine.

The powerfail interrupt is not initiated until processor IPL drops below 30. Critical code sequences can block the powerfail interrupt by raising IPL to 31. Chapter *Power Failure and Recovery* describes power failure and recovery.

4.3 Machine Checks and Corrected Error Interrupts

System hardware can recover from certain error conditions. For some recovered errors, hardware generates a corrected error interrupt to notify VMS of potential problems with one or more hardware blocks. In cases where hardware recovery is impossible, hardware generates a machine check. Software recovery from a condition that causes a machine check may or may not be possible.

Table 4–2 Summary of Hardware Interrupts and Machine Checks

SCB Entry Offset	IPL	Description	Service Routine
600_{16}	22	Interval timer interrupt	EXE$HWCLKINT
620_{16}	20	System corrected error interrupt	EXE$SYSTEM_CORRECTED_ERROR[1]
630_{16}	20	Processor corrected error interrupt	EXE$PROCESSOR_CORRECTED_ERROR[1]
640_{16}	30	Powerfail interrupt	EXE$POWERFAIL
660_{16}	31	System machine check abort	EXE$SYSTEM_MACHINE_CHECK[1]
670_{16}	31	Processor machine check abort	EXE$PROCESSOR_MACHINE_CHECK[1]
680_{16}–$6B0_{16}$	—	Processor-specific interrupts	Processor-specific
800_{16}–$1FF0_{16}$	20–23	I/O interrupts	Device-specific

[1] At the time of this writing, the service routines for corrected error interrupts and machine checks are named differently for each [*censored*] system and are found in different system-specific modules. Efforts are underway to make the routine names and module names consistent. The routine names shown in the table are the names that will eventually be used for all systems. These routines will be included in module [CPU*xxyy*]ERROR_ROUTINES_*xxyy*, where *xx* is the system type designator and *yy* is the processor type designator. Until the changes are implemented, the routines are in the following modules (and have the names shown in the table unless mentioned otherwise):

Module [CPU0202]MCHECK_0202 (SYSTEM_A)
Module [CPU0302]ERROR_ROUTINES_0302 (SYSTEM_C)

EXE$PROC_CORRECTED_ERROR
EXE$SYSTEM_MCHK_ABORT
EXE$PROCESSOR_MCHK_ABORT

Module [CPU0402]CPU_SUPPORT_0402 (SYSTEM_B)

Machine checks are initiated in kernel mode at IPL 31 on the kernel stack. They cannot be disabled, although they can be deferred. A maskable machine check is deferred until processor IPL drops below 31.

Console software reserves an area of physical memory called the logout area to record error- and machine-specific information for corrected error interrupts and machine checks. The logout area is described by fields in the per-CPU slot of the hardware restart parameter block (HWRPB). PALcode records error- and machine-specific information for a particular interrupt or machine check in a piece of the logout area called a logout frame. Before dispatching to the appropriate service routine, PALcode records in R4 the offset to the logout frame from the base of the logout area.

Chapter *Bootstrap Processing* describes the per-CPU slot and the HWRPB. Chapter *Error Handling* summarizes VMS handling of machine checks and corrected error interrupts. For implementation-specific information on the contents of logout frames, see the hardware manual for the appropriate system.

4.3.1 System and Processor Machine Checks

An error condition in hardware external to the processor that cannot be corrected by that hardware or by PALcode results in a system machine check. VMS services this error in EXE$SYSTEM_MACHINE_CHECK. System machine checks are maskable.

Error conditions that result in system machine checks include protocol or parity errors on the processor-memory interconnect and unrecoverable memory errors.

A hardware error condition that occurs within the processor and cannot be corrected by it results in a processor machine check. VMS services this error in EXE$PROCESSOR_MACHINE_CHECK. Processor machine checks may or may not be maskable, depending on the error condition.

Error conditions that result in processor machine checks include processor internal cache errors, translation buffer parity errors, and read accesses to nonexistent local I/O space locations.

4.3.2 System and Processor Corrected Error Interrupts

An error condition that occurs in and is corrected by a hardware block external to the CPU module is reported through a system corrected error interrupt. VMS services this error in EXE$SYSTEM_CORRECTED_ERROR.

An example of such an error interrupt occurs on the SYSTEM_B system when the TURBOchannel interface corrects an error in a direct memory access read operation by applying an error correction code (ECC).

An error condition that occurs in and is corrected by the processor hardware is reported through a processor corrected error interrupt. VMS services this error in EXE$PROCESSOR_CORRECTED_ERROR.

Common examples of processor corrected error interrupts are those generated by the processor when instruction stream and data stream errors are corrected through an ECC.

4.4 Processor-Specific Interrupts

The [*censored*] architecture reserves SCB entries at offsets 680_{16} to $6B0_{16}$ for processor-specific interrupts. For more information, see the appropriate hardware manuals.

4.5 Interval Timer Interrupt

The interval timer enables VMS to keep system time. It requests at least 1,000 interrupts every second; the number varies depending on system implementation. Console software records the frequency of interval timer interrupts in the HWRPB (see Chapter *Bootstrap Processing*).

Interval timer interrupts are serviced by EXE$HWCLKINT, in module TIMESCHDL, at IPL 22. Chapter *Time Support* describes EXE$HWCLKINT in detail.

4.6 I/O Interrupts

The *censored* architecture reserves SCB entries at offsets 800_{16} through $1FF0_{16}$ for I/O interrupts. Interrupts from I/O adapters, I/O widgets, and I/O controllers are all vectored through these SCB offsets; which SCB offset is used for a specific interrupt source depends on the system implementation.

Regardless of SCB offset, all I/O interrupts are serviced by a single interrupt service routine, IO_INTERRUPT, in module IOCINTDISP. IO_INTERRUPT locates and calls the appropriate service routine for the interrupt. Chapter *I/O Processing* describes IO_INTERRUPT and I/O interrupt dispatching.

4.7 Unexpected Interrupts and Exceptions

Module ERRORLOG provides 512 entry points named ERL$VEC*xxxx*, where *xxxx* is the hexadecimal representation of an SCB entry offset. Each of these entry points is another name for ERL$UNEXP, in module ERRORLOG. ERL$UNEXP is the default service routine for all SCB entries. An entry for an interrupt or exception of interest to VMS is reinitialized when the executive image containing its service routine is loaded. ERL$UNEXP logs an error message to indicate that an unexpected interrupt occurred.

Chapter 5

Software Interrupts

And now I see with eye serene
The very pulse of the machine.

William Wordsworth, *She Was a Phantom of Delight*

The [*censored*] architecture provides the capability for kernel mode software to request interrupts at any interrupt priority level (IPL) from 1 to 15. Requested by software rather than hardware, these are called software interrupts. In addition, the architecture provides four interrupts to signal asynchronous system trap (AST) delivery to each of the access modes.

Software interrupts are fundamental to the VMS operating system. Software interrupt service routines perform many of the most important system functions of VMS. These include scheduling (IPL 3), I/O postprocessing (IPL 4), servicing processes' time-dependent requests (IPL 7), and dispatching fork processes (IPLs 6 and 8 to 11). This chapter summarizes how VMS uses software interrupts and discusses fork processing in detail.

5.1 AST Delivery Interrupts

An AST is a mechanism for signaling an asynchronous event to a process. An AST delivery interrupt signals that there is an AST for the current process to execute. An AST interrupt is always requested on behalf of the current process and must be serviced in its context.

On a VAX processor, AST delivery to all modes is initiated through an IPL 2 software interrupt and handled by a common interrupt service routine. On a [*censored*] processor, there are four different AST interrupts, one for each access mode. These interrupts still occur at IPL 2 but are no longer requested through the software interrupt request register (SIRR) and are not dispatched through the IPL 2 software interrupt system control block (SCB) entry.

An AST delivery interrupt service routine delivers an AST by transferring control to the designated AST routine at the specified access mode. Some ASTs are requested by the process, for example, as notification of I/O request completion. Others are queued to the process by VMS as part of normal system operations, such as automatic working set limit adjustment.

Although mentioned in this chapter for completeness, these interrupts are not requested through the [censored] software interrupt mechanism. The details of their request mechanism and delivery are given in Chapter *ASTs*.

5.2 Overview

The VMS executive requests a software interrupt to cause an interrupt service routine to execute and perform its designated function. It does this by executing a CALL_PAL MTPR_SIRR instruction to write a particular IPL into the SIRR processor register. Sometime later, when the interrupt request is granted, the initiate-exception-or-interrupt (IEI) PALcode routine dispatches through the appropriate SCB vector to the software interrupt service routine associated with that IPL. Chapter *Interrupts and Exceptions* describes the processor registers involved in software interrupts and the IEI PALcode routine.

VMS uses software interrupts to schedule operating system functions. Using software interrupts is more efficient than periodically checking to see whether these functions need to be done. VMS assigns IPLs to the different operating system functions, in part as an indication of their relative importance.

VMS also uses specific IPLs and interrupt requests at those IPLs to synchronize access to shared data structures. Chapter *Synchronization Techniques* discusses synchronization through raising IPL.

VMS requests the software interrupt service routines for IPLs 3, 4, 6, 7, 8, and 11 from within a hardware interrupt service routine or another software interrupt service routine. Software interrupts at 12 and 14 are requested only as the result of a person's entering a CPU console command. Although VMS provides for fork dispatching at IPLs 9 and 10, VMS itself makes little or no use of them. VMS does not use the software interrupt vectors for IPLs 1, 2, 5, 13, and 15.

The [censored] architecture constrains software interrupt service routines by providing only one bit to indicate that a software interrupt has been requested at a particular IPL. The service routine is thus unable to determine how many requests for it were outstanding when the interrupt request was granted. As a result, either the software must supply some protocol for determining this number or it must be irrelevant to the execution of the service routine. The scheduling interrupt service routine is an example of a routine that has one function to do, regardless of how many times that function has been requested.

Other interrupt service routines use queues to keep track of their work. Each element in the queue represents a specific item of work for the interrupt service routine and an instance of the interrupt's having been requested.

A queue-driven interrupt service routine generally performs each of the work items in its queue before dismissing the interrupt. It removes an item from the queue, if one is present; processes that item; and tries to remove another. If no item was removed from it because the queue is empty, the interrupt service routine's work is complete; it then exits through a CALL_PAL REI instruction. Such a software interrupt service routine reacts gracefully to a no longer necessary interrupt, one granted when there is no work left for the service routine to do.

5.3 Software Interrupt Service Routines

There is no central monitor routine in VMS that controls the sequence of operating system functions. Instead, any executive thread that identifies the need for a particular function performed within a software interrupt service routine can request the associated interrupt. Scheduling operating system functions as software interrupts eliminates any requirement for polling whether these functions need to be done. It also enables more important functions to interrupt less important ones.

Table 5–1 shows the software interrupt service routine functions and their associated IPLs. In some cases, the assigned IPL only indicates the relative importance of the interrupt, and the interrupt service routine runs primarily at a higher IPL for synchronization. The table also shows the common symbolic names for these IPLs, defined by the macro $IPLDEF.

VMS interprets software interrupts, except the AST delivery and rescheduling interrupts, as systemwide events that are serviced independently of the context of a specific process. These higher IPL software interrupt service routines are said to run in system context. Chapter *System Overview* contains a comparison of system context and process context.

Table 5–1 Software Interrupt Levels Used by the Executive

IPL	IPL Names	Software Interrupt Use
1		Unused
2	IPL$_ASTDEL	Unused[1]
3	IPL$_RESCHED	Rescheduling
4	IPL$_IOPOST	I/O postprocessing
5		Unused
6	IPL$_QUEUEAST	Fork dispatching
7	IPL$_TIMERFORK	Software timer
8	IPL$_SYNCH, IPL$_IOLOCK8, IPL$_SCHED, IPL$_SCS, IPL$_TIMER, IPL$_MMG	Fork dispatching
9	IPL$_IOLOCK9	Fork dispatching
10	IPL$_IOLOCK10	Fork dispatching
11	IPL$_MAILBOX, IPL$_IOLOCK11	Fork dispatching
12		IPC intervention
13		Unused
14		XDELTA
15		Unused

[1] AST interrupts are IPL 2 interrupts; they are distinct, however, from the IPL 2 software interrupt, which is currently unused.

Some software interrupt service routines perform the same functions every time they execute, while others perform more varied functions. The rescheduling interrupt service routine, for example, takes the current process out of execution, selects another one to run, and places it into execution. The I/O postprocessing interrupt service routine has a specific function to perform but is data-driven by the I/O request packets (IRPs) in its work queue. A fork dispatching interrupt exists solely to dispatch to system routines running as fork processes, but which routines and fork processes execute varies with system operation.

The software interrupts used by VMS are described briefly in the following sections. Some are described at more length in subsequent chapters. The following sections are in order by interrupt level except that the fork dispatching interrupts are discussed last.

The vectors for unused software interrupts all point to EXE$HALT, which contains a `CALL_PAL HALT` instruction. Consequently, unexpected software interrupts result in a processor halt.

5.3.1 Rescheduling Interrupt (IPL 3)

The executive requests a rescheduling interrupt at IPL 3 whenever a resident process of high enough priority to preempt the current process becomes computable.

The IPL 3 interrupt service routine, SCH$INTERRUPT in module SCHEDULER, begins execution at IPL 3 on the kernel stack of the current process. It acquires the SCHED spinlock, raising IPL to IPL$_SCHED, and changes the state of the current process from current (CUR) to computable (COM). It calls SCH$FIND_NEXT_PROC, in module SCHED_ROUTINES, to select the highest priority resident computable process and executes a `CALL_PAL SWPCTX` instruction, switching from the context of the previous process to that of the new process. The new process's scheduling state is changed to CUR.

Many of the events that make a process computable occur as part of servicing software interrupts between IPL$_IOPOST and IPL$_SCHED. That the scheduler database is modified from these software interrupts has the following implications:

- SCH$INTERRUPT must raise IPL to IPL$_SCHED and acquire the SCHED spinlock to block any other accesses to the scheduler database while it takes one process out of execution and selects another one to run.
- The IPL 3 interrupt can be requested a number of times before it is granted. The number of times the interrupt has been requested is not recorded but is irrelevant, since the interrupt service routine always does the same task.
- When the IPL 3 interrupt is granted, all events that might affect the choice of which process to run have been serviced. That is, the higher priority software interrupt service routines that affect the scheduler database have completed their work. Thus, the interrupt service routine can make the best possible choice at the time it blocks further alterations to the database.

Chapter *Scheduling* discusses the scheduler database, events that affect the scheduler database, and the rescheduling interrupt.

5.3.2 I/O Postprocessing (IPL 4)

When a device driver has completed an I/O request, it invokes a routine that places the IRP associated with the request at the tail of the I/O postprocessing queue. If the queue was empty, the routine requests a software interrupt at IPL$_IOPOST (IPL 4). If the queue was not empty, a software interrupt has already been requested; requesting another would impose additional overhead and serve no purpose.

Early versions of VAX/VMS had one I/O postprocessing queue. As part of support for symmetric multiprocessing (SMP), VAX/VMS Version 5 provides a per-CPU queue for each member of the system and a systemwide I/O postprocessing queue, to which most IRPs are queued. An IRP for a request completed in process context (that is, by a driver's function decision table action routine) is typically queued to the per-CPU queue. The VAX/VMS I/O postprocessing interrupt service routine, running on a uniprocessor or on the primary processor of an SMP system, services the systemwide queue and its own per-CPU queue. The VAX/VMS I/O postprocessing interrupt service routine, running on a secondary processor, services only that processor's per-CPU queue.

Although [*censored*] VMS does not support SMP in this release, it is based upon VAX/VMS Version 5.4 and retains support for multiple I/O postprocessing queues: a systemwide queue and a single per-CPU queue.

The I/O postprocessing interrupt software routine, IOC$IOPOST in module IOCIOPOST, removes each IRP in turn from the beginning of the systemwide queue and processes it. The details of the processing vary with the type of IRP. For example, IOC$IOPOST distinguishes between VMS buffered and direct I/O requests. When a direct I/O request completes, IOC$IOPOST unlocks the buffer pages from memory. When a buffered output request completes, IOC$IOPOST deallocates the buffer to nonpaged pool and returns process byte count quota.

IOC$IOPOST then services the single per-CPU I/O postprocessing queue. After it processes all IRPs in the queue, it dismisses the interrupt with an REI statement. Example 5–1, a slightly simplified extract from module IOCIOPOST, shows this sequence.

Example 5–1 IOC$IOPOST Interrupt Service Routine Extract

```
;IOC$IOPOST::                          ;I/O postprocessing interrupt
IOPOST:
        EVAX_MFPR_PRBR                 ;Get per-CPU data address into R0
        CMPL    CPU$L_PHY_CPUID(R0),SMP$GL_PRIMID ;Are we the primary?
        BNEQ    5$                     ;If NEQ, no
        TSTL    IOC$GQ_POSTIQ          ;Is systemwide queue empty?
        BEQL    5$                     ;Branch if yes, service per-CPU queue
        $REMQHI IOC$GQ_POSTIQ,R5       ;Get IRP from systemwide queue
        BVC     60$                    ;If VC, got one
5$:     REMQUE  @CPU$L_PSFL(R0),R5     ;Get IRP from per-CPU queue
        BVC     60$                    ;If VC, got one
        REI                            ;No more IRPs to postprocess.  Exit
60$:    .                              ;Postprocess this
        .                              ; I/O request packet
        BRW     IOPOST                 ;Get next I/O request packet
```

Chapter *I/O System Services* contains further general information about I/O postprocessing, and Chapter *Paging Dynamics* gives specific information about postprocessing of memory management requests.

5.3.3 Software Timer (IPL 7)

VMS includes both a hardware interrupt service routine to service the hardware clock and a software timer interrupt service routine. Together these routines service time-dependent requests. Chapter *Time Support* describes them in detail; this section summarizes some of their interaction.

The hardware interrupt service routine is EXE$HWCLKINT in module TIMESCHDL. It runs in response to a hardware clock interrupt at IPL 22. Some of its duties are to update the system time, perform CPU time accounting, check for quantum expiration of the current process, and check whether the first timer queue entry (TQE) has come due. If the current process has run out of quantum or if the first TQE has come due, EXE$HWCLKINT requests an IPL$_TIMERFORK (IPL 7) interrupt.

A TQE describes a time-dependent request, which is typically made through the Schedule Wakeup ($SCHDWK) or Set Timer ($SETIMR) system service. The queue of TQEs is kept ordered by expiration time, with the soonest due first. Quantum-end processing and TQE servicing require lengthier execution than is appropriate at high IPL and require modification to the scheduler database, which is synchronized at IPL$_SCHED.

The IPL$_TIMERFORK interrupt service routine, EXE$SWTIMINT in module TIMESCHDL, checks whether the current process's quantum has expired. If it has, EXE$SWTIMINT invokes the routine that performs quantum-end processing.

EXE$SWTIMINT acquires the TIMER and HWCLK spinlocks, raising IPL to IPL$_HWCLK, to synchronize its access to the queue of TQEs and, in particular, the first TQE. It removes the first TQE if its expiration time is the same as or earlier than the current system time. It releases the two spinlocks, lowering IPL to IPL$_TIMER (IPL 8). EXE$SWTIMINT then processes the TQE.

It reacquires the TIMER and HWCLK spinlocks and checks the TQE that is now first in the queue. EXE$SWTIMINT continues in this manner until it reaches a TQE that has not yet expired. It then releases the two spinlocks and dismisses the interrupt, leaving unexpired TQEs in the queue.

5.3.4 IPC (IPL 12)

The IPL 12 interrupt is only requested by a person depositing 12 into the software interrupt request register at the CPU console terminal. The IPL 12 interrupt service routine, EXE$IPCONTROL in module IPCONTROL, facilitates certain types of human intervention when the system might otherwise have to be crashed.

When the IPL 12 interrupt request is granted, the interrupt service routine prompts on the console for human input with the following text: `IPC>`. (IPC is a shortened form of IPL C, where C_{16} is 12.) The IPL 12 interrupt service routine accepts the following commands:

Command	Meaning
C *ddcu:*	Cancel mount verification in progress
X	Activate XDELTA (if it is resident)
CTRL/Z	Return the system to normal operation

The C command is issued with a device specification to cancel mount verification on the specified disk or tape. Mount verification is a mechanism that enables the system to recover gracefully from certain kinds of transient device failures by stalling I/O requests to a device while it is offline or inaccessible. If the device comes back on line, the system confirms that this is the same device as was previously mounted and resumes normal I/O processing on the volume. If SYSGEN parameter MVTIMEOUT seconds elapse before a disk comes back on line, mount verification times out and the system aborts I/O requests in progress to that disk. For a tape, the SYSGEN parameter TAPE_MVTIMEOUT specifies the length of the mount verification timeout period.

While a device is in a state of mount verification in progress, all users' I/O requests to it are stalled until the mount verification times out or the device comes back on line. An impatient user can type CTRL/C or CTRL/Y and STOP to abort the image and cancel its I/O requests. However, the user cannot cancel any I/O request the file system might have made on the user's behalf, and subsequent file system activity in the process will be blocked until mount verification times out or is canceled.

Therefore, if the device failure is known to be permanent, it may be appropriate to cancel mount verification before the mount verification timeout period has elapsed. In most cases, the DISMOUNT/ABORT command is the preferred way to cancel mount verification. (See the *VMS DCL Dictionary* for further information on this command.) However, if the state of the system prevents that command from being entered, the C command to the IPL 12 interrupt service routine may be entered instead.

For additional information on mount verification, see the *Guide to Maintaining a VMS System*.

In response to an X command, EXE$IPCONTROL invokes INI$BRK to activate XDELTA, as described in Section 5.3.5. Note, however, that this method is not commonly used because XDELTA can be activated directly through a higher priority IPL 14 interrupt.

In response to CTRL/Z, EXE$IPCONTROL exits, dismissing the IPL 12 interrupt with a `CALL_PAL REI` instruction.

5.3.5 XDELTA (IPL 14)

XDELTA, the executive debugger, can optionally be loaded at system initialization. If XDELTA is resident, the SCB vector for the breakpoint exception points to a service routine within XDELTA. XDELTA remains quiescent, transferring control to the usual exception service routine when breakpoints occur, until a breakpoint is executed whose address is contained in XDELTA's breakpoint table. Initially, the only such breakpoint is at routine INI$BRK, in module SYSTEM_ROUTINES.

When such a breakpoint instruction is executed, XDELTA accepts command input from the CPU console terminal. User-entered commands can include setting other breakpoints, setting single-step mode, and examining system space. Often programmers debugging kernel mode code that runs above IPL 0, such as a device driver, invoke INI$BRK from their code to activate XDELTA. The *VMS Delta/XDelta Utility Manual* provides further information about XDELTA (and DELTA) commands.

VMS uses the IPL 14 software interrupt to enable a person to activate XDELTA by depositing 14 in the software interrupt request register at the CPU console terminal. The interrupt service routine to activate XDELTA is INI$MASTERWAKE, in module SYSTEM_ROUTINES. This interrupt service routine consists of a call to INI$BRK and an REI instruction. When XDELTA is entered through its BPT exception service routine, it raises IPL and executes at IPL 31. When XDELTA is dismissed through a ;P or ;G command, it returns to INI$MASTERWAKE, which dismisses the interrupt.

When XDELTA is not resident, the instruction at INI$BRK is a NOP rather than a CALL_PAL BPT. Thus, a system without XDELTA reacts gracefully to an XDELTA interrupt or a JSB to INI$BRK.

5.4 Fork Processing

Five software interrupts (IPLs 6 and 8 to 11) exist solely to dispatch to fork processes. Although called a process, a fork process bears little resemblance to a normal process. A fork process is simply a routine invoked by one of these interrupt service routines. The routine's code and any data it accesses must be in nonpageable system space. A fork process and its context are described by a fork block (FKB).

Each of the interrupt service routines has its own work queue of fork blocks. When one of these interrupts is granted, the interrupt service routine removes from its queue each fork block in turn and dispatches to the fork process it describes. The fork process mechanism essentially creates a list of routines to be executed based on current system needs.

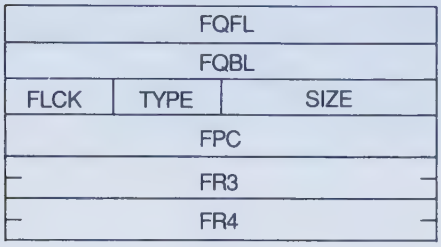
The following sections describe fork process data structures and service routines in more detail.

5.4.1 Fork Process Data Structures

A fork block describes a routine to be called by a fork dispatching interrupt service routine and some context for that routine. The macro $FKBDEF defines symbolic names for the fields in a fork block. A minimal fork block, shown in Figure 5–1, includes the procedure value of the fork process entry point (FKB$L_FPC) and the contents of R3 and R4. (Note that the field name FKB$L_FPC stands for fork program counter (PC) on VAX/VMS. The field name is unchanged for [*censored*] VMS, although the exact meaning of its contents has changed.) The first two longwords of a fork block link it into a queue. The fields FKB$W_SIZE and FKB$B_TYPE complete the standard dynamic data structure header.

The field FKB$B_FLCK identifies the spinlock associated with the fork process. It is an index into a table of static spinlocks, pointed to by SMP$AR_SPNLKVEC, and also into a table of spinlock IPLs, at SMP$AL_IPLVEC. The indexed value in the spinlock IPL table specifies into which fork block queue the fork block is inserted and at what IPL its routine will run. (In contrast, on VAX/VMS the field FKB$B_FLCK, also known as FKB$B_FIPL, can contain an actual fork IPL.) Chapter *Synchronization Techniques* describes spinlocks in detail.

Figure 5–1 Layout of a Fork Block



A fork block must be in nonpageable system space. Most often, it is part of a larger data structure, such as a unit control block or class driver request packet, which contains additional data. The combination of standard fork block fields, additional fork block data, and the routine that is to be executed is called a fork process.

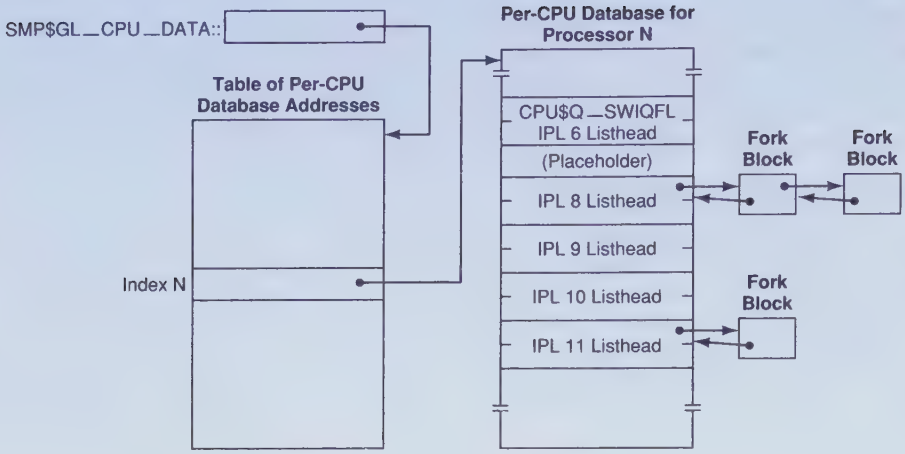
Figure 5–2 shows the array of fork block queue listheads. In VAX/VMS Version 5 the array is part of the per-CPU database so that each member of an SMP system has its own fork block queues. In [*censored*] VMS the array remains in the per-CPU database, although only one CPU is supported in this release. The listheads of these queues are ordered in an array that includes a placeholder listhead for IPL 7. Since the IPL 7 interrupt is serviced by the software timer routine, there is no fork process dispatching at IPL 7. However, having the placeholder listhead simplifies the fork process creation code.

5.4.2 Reasons for Creating a Fork Process

Fork processing exists, in part, so that device drivers do not have to execute at high IPLs for long periods of time, blocking other device interrupts. Device interrupt service routines run at device IPLs between 20 and 23. Often, servicing an interrupt requires lengthy processing but not high-IPL execution. Typically, a device interrupt service routine switches to a lower IPL as soon as possible. However, it may not simply lower IPL directly; that could interfere with the synchronization of code already running at the lower IPL. Instead, it creates a fork process that will run at the lower IPL when its turn comes.

A driver or any high-IPL thread of execution might also create a fork process at a lower IPL to access a system database synchronized at that lower IPL, for example, if the driver needed to queue an AST to a process. Another example is the routine that allocates nonpaged pool. It can be invoked from process context

Figure 5–2 Fork Block Queues



code and from interrupt threads of execution at IPLs up to 11. If the routine determines that pool must be expanded before any can be allocated, but IPL is above IPL$_MMG or the processor holds a higher ranking spinlock than MMG, the routine creates an IPL 6 fork process to perform the expansion. Chapter *Pool Management* gives more information on pool allocation.

5.4.3 Creating a Fork Process

To fork, a driver invokes routine EXE$PRIMITIVE_IOFORK, EXE$PRIMITIVE_FORK, or EXE$QUEUE_FORK, in module FORKCNTRL, specifying the address of the fork block, the contents of R3 and R4, and the procedure value of the fork process entry point.

Generally a driver routine accomplishes this by invoking the macro FORK or IOFORK, either of which generates code that performs the following steps:

1. It stores the procedure value of the fork process entry point in the fork block field FKB$L_FPC.
2. It invokes EXE$PRIMITIVE_FORK or EXE$PRIMITIVE_IOFORK.
3. When either of those routines returns, it returns to code generated by the macro that returns to the invoker of the driver routine (the default) or transfers to a specified place within the driver routine containing the macro.

Example 5–2 shows the VAX MACRO statements generated by a simple invocation of the FORK macro.

EXE$PRIMITIVE_IOFORK clears a bit to disable an I/O timeout on the device and continues in the EXE$PRIMITIVE_FORK routine.

Example 5–2 FORK Macro and Generated Code

```
;  The macro invocation
        FORK
;  The generated code
        $FKBDEF
        MOVAB   L1,FKB$L_FPC(R5)
        JSB     G^EXE$PRIMITIVE_FORK
        RSB
L1:     .JSB_ENTRY  -
           INPUT=<R3,R4,R5>,SCRATCH=<R0,R1,R2>
```

Example 5–3 EXE$PRIMITIVE_FORK Routine Extract

```
        UNIVERSAL_JSB   EXE$PRIMITIVE_FORK,INPUT=<R3,R4,R5>,OUTPUT=<R3,R4>
;EXE$PRIMITIVE_FORK::
        EVAX_STQ-                       ;Save R3 and R4 in fork block
                R3,FKB$Q_FR3(R5)        ;Use a built-in to store all
        EVAX_STQ-                       ; 64 bits of the register
                R4,FKB$Q_FR4(R5)
                                        ;Fall through into EXE$QUEUE_FORK
        UNIVERSAL_JSB   EXE$QUEUE_FORK,INPUT=<R5>,OUTPUT=<R3,R4>
;EXE$QUEUE_FORK::
        MOVZBL  FKB$B_FLCK(R5),R4       ;Get fork lock index
        MOVL    G^SMP$AL_IPLVEC[R4],R4  ;Get fork IPL from spinlock database
        FIND_CPU_DATA R3                ;Get base of CPU data area
        ASSUME  FQH_SIZE EQ 8           ;Assumption required for use of MOVAQ
        MOVAQ   -                       ;Get address of fork queue listhead
                CPU$Q_SWIQFL-<6*FQH_SIZE>(R3)[R4],R3
        INSQUE  (R5),@4(R3)             ;Insert fork block in fork queue
        BNEQ    10$                     ;If queue already populated
                                        ; avoid extra interrupt
        SOFTINT R4                      ;Request software interrupt
10$:    RSB                             ;Return
```

EXE$PRIMITIVE_FORK stores R3 and R4 in the fork block. At subentry point EXE$QUEUE_FORK, it converts the spinlock index in FKB$B_FLCK to an IPL by indexing the array at SMP$AL_IPLVEC with it. EXE$PRIMITIVE_FORK inserts the fork block at the tail of the fork block queue for that IPL and requests a software interrupt at that IPL if the queue was empty.

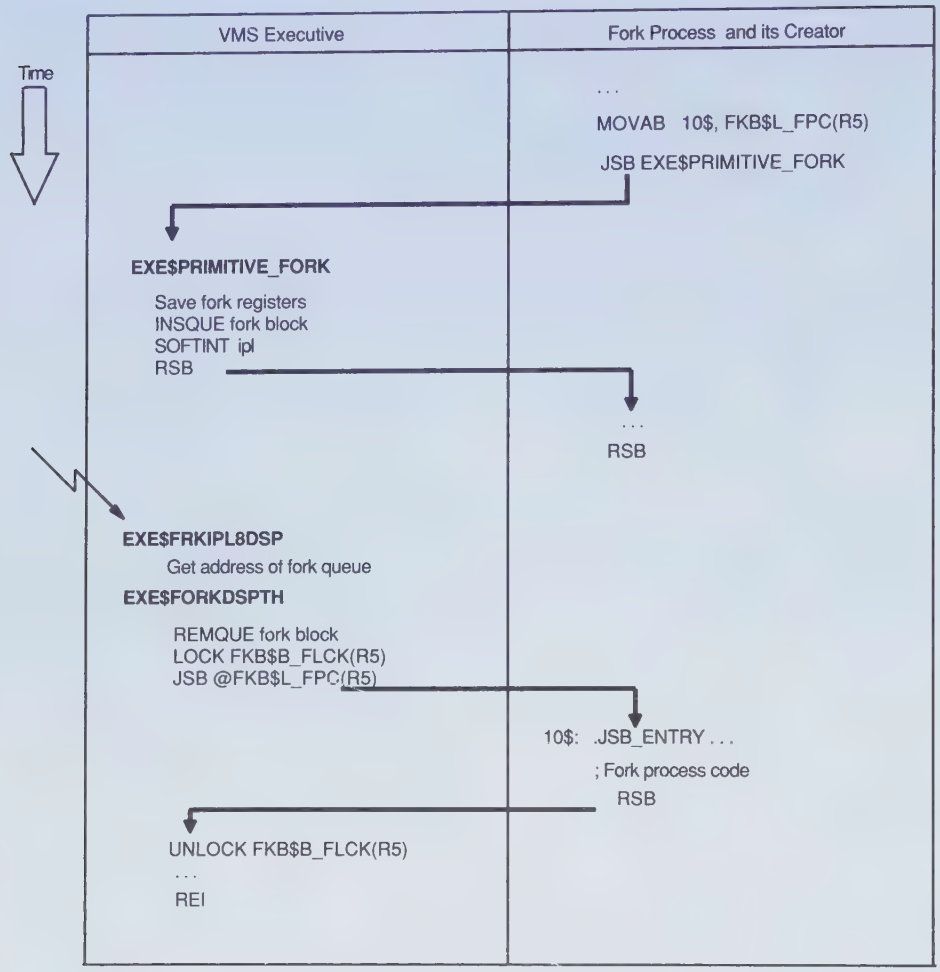
EXE$PRIMITIVE_FORK then returns to the macro-generated code, which returns as previously described.

The instructions in EXE$PRIMITIVE_FORK that perform these functions are listed in Example 5–3. Figure 5–3 shows the general flow of the creation of a fork process and dispatch into it.

When IPL drops below that of the requested fork interrupt, the fork dispatching interrupt will be granted and serviced.

The steps to create a fork process are different from those on VAX/VMS. A VAX/VMS driver executes a JSB to EXE$FORK, which pops the return address off the stack and stores it in FKB$L_FPC as the fork process entry point. When EXE$FORK returns, it returns to the invoker of the driver routine; that is, it returns to "its caller's caller." [*Censored*] VMS does not support that form of return and instead requires nested returns.

Figure 5–3 Creating and Dispatching to a Fork Process



Requiring explicit control of the stack, the caller's caller form of return is not feasible for code written in anything but an assembler language. A typical VAX MACRO routine compiled for [*censored*] execution, for example, begins with compiler-generated register saves and ends with compiler-generated register restores. To ensure that saved registers and the state of the stack are restored, a routine must execute its own return code.

For a fork process routine that must maintain context on the stack across stalls, [*censored*] VMS provides a mechanism called a kernel process, described in Section 5.5.

5.4.4 Dispatching a Fork Process

When a fork interrupt is granted, the IEI PALcode routine dispatches to its interrupt service routine. Each fork IPL has a unique interrupt service routine that performs setup for common fork dispatching code. The interrupt service routine stores the offset of the corresponding fork queue listhead in R6 and branches to the common fork dispatching code. The interrupt service routines for IPLs 6 and 8 and the common fork dispatching code, EXE$FORKDSPTH, are listed in Example 5–4. These routines are in module FORKCNTRL.

EXE$FORKDSPTH loads R6 with the address of the fork block queue specified by the sum of R6 and the address of the per-CPU database for this processor. It removes the first fork block from the queue. It loads R3 and R4 from the fork block. It acquires the spinlock whose index is in FKB$B_FLCK. It then initiates the fork process by invoking the routine whose procedure value address is in FKB$L_FPC. When the fork process returns, EXE$FORKDSPTH releases the spinlock whose index is in FKB$B_FLCK. The fork process itself must not release the spinlock before returning; if it does, EXE$FORKDSPTH's attempted release will cause the system to crash. Note that even though all the fork processes EXE$FORKDSPTH initiates as the result of a particular interrupt execute at the same IPL, they do not necessarily acquire the same fork lock; each fork block identifies the fork lock the associated fork process must acquire.

EXE$FORKDSPTH then removes the next fork block and processes it in the same manner as the first. The removal and processing continue until the queue is empty, when the dispatcher dismisses the interrupt with an `REI` statement. Note that, to improve performance, EXE$FORKDSPTH detects removal of the last entry in the queue and avoids a subsequent fruitless `REMQUE` by dispatching the last entry in a separate code path.

Figure 5–3 shows a simplified form of this flow.

Since a fork process routine runs in system context as the result of an interrupt above IPL 2, it cannot rely on any particular process context. It may access no per-process address space except the stack on which it runs (which can be a kernel stack in some process's P1 space). Its code and data must be in nonpageable system space; it must not incur page faults, execute change mode instructions, or incur any exceptions that are dispatched to user-defined condition handlers (see Chapter *Condition Handling*). While a fork process is executing, it may use R0 through R5 and, if saved and restored, the other general registers. A fork process may also use the stack. However, when a fork process returns control to the fork dispatcher, the stack must be in the same state as when the fork process was entered.

5.4.5 Stalling a Fork Process

A fork process can be stalled for various reasons and have to wait. When a fork process waits, its context is saved in the FKB: R3, R4, and the procedure value corresponding to where it should be reentered. The FKB is then placed in a queue of FKBs.

Example 5–4 Fork Dispatching Interrupt Service Routine Extract

```
EXE$FRKIPL6DSP:                         ;Fork IPL 6 entry point
        CLRL    R6                      ;Get offset to fork queue listhead
        BRB     EXE$FORKDSPTH           ;Branch to common code
      .
      .
EXE$FRKIPL8DSP:                         ;Fork IPL 8 entry point
        MOVZBL  #<<8-6>*FQH_SIZE>,R6    ;Get offset to fork queue listhead
;
; Drop through to common code
;
EXE$FORKDSPTH:                          ;Software interrupt fork dispatcher
        FIND_CPU_DATA   R7              ;Get base of CPU-specific database
        MOVAB   CPU$Q_SWIQFL(R7)[R6],R6 ;Get address of fork queue listhead
10$:    REMQUE  @(R6),R5                ;Remove next entry from fork queue
        BNEQ    20$                     ;Branch if more follow the remqued entry
        BVS     90$                     ;Branch if none to remove
        CLRL    R6                      ;Otherwise, this is last, so clear R6
;
; Obtain fork level spinlock and dispatch to fork routine with
;
;   R0,R1,R2    available as scratch registers
;   R3,R4       restored from the FORK block
;   R5          pointing to the FORK block
;
; On return from the fork routine, release the spinlock
;
20$:    EVAX_LDQ-               ;Restore R3 and R4 for fork routine
                FKB$Q_FR3(R5),R3
        EVAX_LDQ-               ;Use built-in to load all 64 bits
                FKB$Q_FR4(R5),R4
        MOVZBL  FKB$B_FLCK(R5),R7       ;Get fork lock number
        FORKLOCK -                      ;Acquire spinlock
                LOCK    =R7,-           ; using fork lock index
                PRESERVE=NO             ; destroying R0
        JSB     @FKB$L_FPC(R5)          ;Execute fork routine
        FORKUNLOCK -                    ;Release the spinlock
                LOCK    =R7,-           ; using fork lock index
                PRESERVE=NO             ; destroying R0
        TSTL    R6                      ;If this was not the last entry
        BNEQ    10$                     ; then go back and REMQUE next

90$:    REI                             ;Restore registers, dismiss interrupt
```

An example of such a wait is a driver fork process that tries to obtain exclusive access to its device's controller. The fork process is stalled until another fork process that has previously obtained exclusive access releases the controller. The routine called to release the controller restores the context of the waiting fork process so that it can repeat its attempt to obtain the controller. Note that all fork processes that can stall waiting for a particular resource must use the same spinlock.

VMS also implements a "fork and wait" wakeup mechanism so that a fork process can stall itself for a short while and be awakened automatically. To fork and wait, a fork process invokes the macro FORK_WAIT.

The disk and tape class drivers use this mechanism after an unsuccessful attempt to allocate nonpaged pool, assuming that nonpaged pool will become available. When the fork process is reentered, it repeats its attempt to allocate nonpaged pool. In this example, the fork and wait mechanism is used in lieu of nonpaged pool availability reporting, the mechanism used by full processes (see Chapters *Scheduling* and *Pool Management*).

The FORK_WAIT macro generates code that performs the following steps:

1. It stores the procedure value of the fork process entry point in the fork block field FKB$L_FPC.
2. It invokes EXE$PRIMITIVE_FORK_WAIT, in module FORKCNTRL.
3. EXE$PRIMITIVE_FORK_WAIT returns to code generated by the macro that returns to the invoker of the driver routine (the default) or transfers to a specified place within the driver routine containing the macro. Typically the macro generates a return to the fork dispatcher, which releases the fork lock and dispatches the next fork process.

EXE$PRIMITIVE_FORK_WAIT saves the fork process's context in the fork block. Raising IPL to 31, it then acquires the MEGA spinlock, which serializes access to the systemwide fork and wait queue, and inserts the fork block at the tail of the queue. EXE$PRIMITIVE_FORK_WAIT then releases the MEGA spinlock, restoring the IPL at entry, and returns to its invoker.

The fork and wait queue is serviced once a second by the routine EXE$TIMEOUT, in module TIMESCHDL. Thus, on average, the fork process waits for half a second. EXE$TIMEOUT acquires the MEGA spinlock to serialize its access to the fork and wait queue. It copies the queue listhead, whose address is in EXE$AR_FORK_WAIT_QUEUE, initializes the listhead to represent an empty queue, and releases the MEGA spinlock. EXE$TIMEOUT removes each fork block in turn from its copy of the listhead and restores its R3 and R4. EXE$TIMEOUT acquires the spinlock whose index is in FKB$B_FLCK, raising to the IPL associated with the spinlock. EXE$TIMEOUT then dispatches to the fork process. When the fork process returns, EXE$TIMEOUT releases the spinlock. When the copied listhead is empty, EXE$TIMEOUT is done servicing the queue and continues with other processing.

Part of the restoration of fork process context involves changing IPL from IPL$_TIMER to the IPL indirectly specified by FKB$B_FLCK. Because lowering IPL would violate the interrupt nesting scheme, use of the fork and wait mechanism is limited to fork processes with fork IPLs at or above IPL$_TIMER.

Chapter *Time Support* contains further information about EXE$TIMEOUT.

5.4.6 Use of Fork IPLs

There are five different fork IPLs; three are used by most device drivers supplied as part of VMS:

- IPL 6 is for drivers that support attention ASTs or that otherwise need access to the scheduler database or any other data synchronized at a lower IPL or with a lower ranked spinlock than that of the driver fork process. Chapter

Synchronization Techniques discusses the need for IPL 6 fork processing in somewhat more detail.

- IPL 11 is for the mailbox driver. The mailbox driver runs at the highest fork IPL so that any driver fork process can write mailbox messages without the need to fork to a lower IPL. A driver fork process, for example, may write a device-offline message to the OPCOM process's mailbox.
- IPL 8 is the most common driver fork IPL.

The following considerations affect the choice of fork lock and IPL for any particular driver:

- Higher fork IPLs are serviced first.
- All device drivers on a particular bus or adapter competing for bus or adapter resources, such as map registers, must use the same fork lock and thus run at the same fork IPL. When one fork process deallocates a resource for which another fork process is waiting, the waiting fork process is resumed. The resource routine assumes that both fork processes use the same spinlock; it does not release the deallocator's fork lock, nor does it acquire the waiting fork process's fork lock. If the two fork processes were to use different fork locks, the waiting fork process would fail to synchronize its accesses to data structures protected by its fork lock.

  Any direct memory access (DMA) drivers servicing devices on a bus or adapter one of whose DMA devices is serviced by a VMS driver must use the IOLOCK8 spinlock.
- All device drivers that compete for exclusive access to the same controller must use the same fork lock for the reason just described.
- All system communication services (SCS) class and port drivers must use the IOLOCK8 spinlock, partly for the reason previously described and additionally because SCS routines invoked by these drivers assume that the IOLOCK8 spinlock (also known as the SCS spinlock) is held.
- A disk driver must use the IOLOCK8 spinlock in order to synchronize mount verification.
- A driver that accesses a systemwide database synchronized with a spinlock whose associated IPL is IPL$_SYNCH can do so from fork level if the following conditions are met:
  - — Its fork IPL is 8, the value of IPL$_SYNCH.
  - — The driver's fork lock is either the same one that synchronizes the database of interest or is one of lower rank so that the fork process can acquire the needed spinlock.

5.5 Kernel Processes

As previously described, VMS requests software interrupts to schedule operating system functions. Some software interrupt service routines perform the same function and execute the same routines each time they execute. Others, most notably the fork dispatchers, dispatch to the routines specified by the data structures in their work queues. Various executive components that identify the need to execute a particular system context thread of execution at a particular IPL create a work queue data structure to represent the thread. Creating a fork process is the most common way to do this. Other possibilities are timer queue entry system subroutines dispatched by EXE$SWTIMINT and end action routines dispatched by IOC$IOPOST.

The essential characteristics of such a system context thread of execution are as follows:

- The constraint that it access only system space (and, on *censored* VMS, possibly a P1 space kernel stack)
- A particular routine to execute
- A data structure containing information for the routine
- Execution at elevated IPL (commonly 6 or 8 through 11, the fork IPLs)

Typically, such a system context thread of execution is short-lived; once dequeued and initiated, it completes its task immediately and exits. Sometimes, however, a thread cannot complete its task immediately and instead must stall itself to restart later. Such a system context thread of execution can use the stack on which it runs but cannot leave information on the stack between a stall and a restart.

The kernel process mechanism, new with *censored* VMS, enables a system context thread of execution to run on its own private stack. While a kernel process is stalled, it can leave execution state on the stack, such as nested stack frames and saved registers. This ability to save execution state across a stall is the primary motivation for kernel processes. It simplifies driver algorithms that are naturally expressed as nested subroutine calls and that would otherwise require complex state descriptions. Also, this ability is a prerequisite to supporting device drivers written in a high-level language.

To run as a kernel process, a system context thread of execution, such as a fork process, calls a set of VMS-provided routines that preserve register context and switch stacks. Entry into and exit from a kernel process always involves a stack switch:

- At initiation, a switch from the current kernel stack to that of the kernel process
- At a stall, a switch from the kernel process's stack to the one current when the kernel process was entered
- At restart, a switch from the current kernel stack to that of the kernel process
- At termination, a switch from the kernel process's stack to the one current when the kernel process was most recently entered

Basically, calling these VMS-provided routines maintains the environment of a kernel process. Any system context thread of execution that calls these routines to run on its own stack is a kernel process. That the thread of execution runs on a private stack is transparent to the rest of the system. The thread of execution must be entered through a standard mechanism and then call the appropriate VMS routine to begin or resume execution as a kernel process on a private stack. Once executing as a kernel process, in order to stall, the thread must call a routine that can switch stacks and then save the thread's state in such a way that it can restart when the stall ends. Currently, VMS provides routines that transform driver fork processes back and forth into kernel processes; only driver fork processes currently use the kernel process mechanism.

Section 5.5.1 describes the data structures associated with kernel processes, and Section 5.5.2, the routines that support them. Section 5.5.3 discusses an example kernel process flow.

5.5.1 Kernel Process Data Structures

Two data structures are associated with each kernel process:

- A kernel process block (KPB) describes the context and state of a kernel process.
- A stack records the current state of the kernel process's execution.

Typically, the data structures are created when a kernel process is initiated and deleted when it ends. However, it is possible for the data structures to be preallocated and used by many kernel processes. In particular, VMS preallocates a set of kernel process data structures for each possible concurrent modified page write. Chapter *Paging Dynamics* discusses the modified page writer.

A KPB is a nonpaged pool data structure of type DYN$C_MISC and subtype DYN$C_KPB. The data structure is variable-length, consisting of one mandatory and several optional areas. The KPBs created for VMS executive use (referred to in code as VMS executive software type KPBs, or VEST KPBs) have a standard form, shown in Figure 5–4.

The macro $KPBDEF defines symbolic names for the fields in a KPB. KPBIW_SIZE, KPBIB_TYPE, and KPB$IB_SUBTYPE are the standard dynamic data structure header fields.

The areas of a KPB are

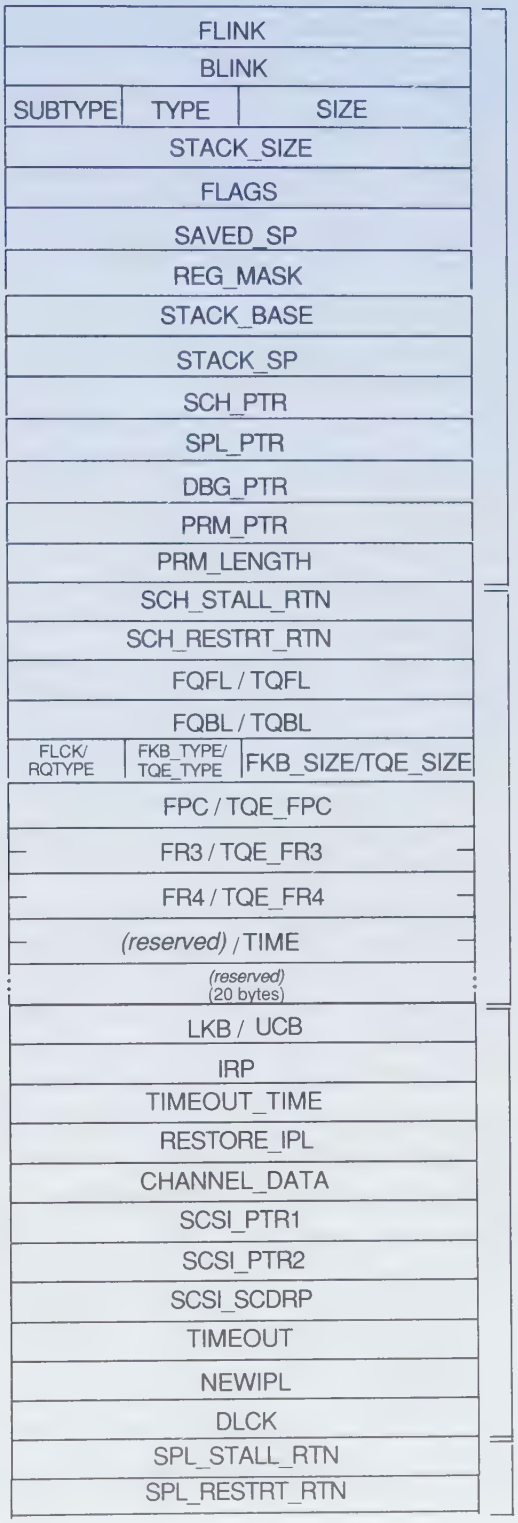
- The base area, present in all KPBs
- The scheduling and executive special parameters areas, present in all KPBs
- A spinlock area, present in VMS executive KPBs
- A debug area, not used in any production software
- A general parameters area, not used by VMS executive KPBs

All but the general parameters area have a fixed size; KPB$IS_PRM_LENGTH contains the size in bytes of the general parameters area, if it is present. Fields

Figure 5–4 Layout of a VMS Executive Kernel Process Block

KPB FLAGS

Bit	Name	Meaning
0	VALID	Valid
1	ACTIVE	In active use
2	VEST	VMS executive format
3	DELETING	Deletion pending
4	SCHED	Scheduling and special parameters area present
5	SPLOCK	Spinlock area present
6	DEBUG	Debug area present
7	PARAM	General parameters area present
8	DEALLOC_AT_END	Deallocate KPB when kernel process ends



KPBPS_SPL_PTR, KPBPS_DBG_PTR, and KPB$PS_PRM_PTR point to optional areas; each area has a bit in KPB$IS_FLAGS, which, when set, indicates the area's presence in the KPB. Because each KPB has scheduling and executive special parameters areas whose fields can be accessed through fixed offsets, KPB$PS_SCH_PTR is not used. Currently all KPBs have the standard form shown in Figure 5–4.

KPBIS_STACK_SIZE, KPBPS_STACK_BASE, and KPB$PS_STACK_SP, shown in Figure 5–5, describe the kernel process's stack. KPB$PS_SAVED_SP contains the stack pointer on the stack current when the kernel process was initiated or restarted. That pointer is restored when the kernel process stalls or terminates.

The scheduling area can contain either a fork block (see Figure 5–1) or a timer queue entry (see Chapter *Time Support*). To use the fork block, the code that creates a kernel process must initialize KPB$IB_FLCK with a spinlock index.

KPB$PS_SCH_STALL_RTN and KPB$PS_SCH_RESTRT_RTN contain procedure values of routines to be called when a kernel process is stalled or restarted. The scheduling stall routine takes whatever steps are necessary to guarantee eventual rescheduling of the kernel process. In current use, these steps include saving fork process context. The scheduling restart routine is currently unused.

The executive special parameters area contains fields to support device driver fork processes.

KPB$PS_SPL_STALL_RTN and KPB$PS_SPL_RESTRT_RTN can contain procedure values of routines to be called when a kernel process is stalled or restarted. The routines release or reacquire spinlocks needed by the kernel process. These routines are currently not necessary: the fork dispatcher acquires and releases the fork spinlock for a kernel process that is entered as a fork routine. However, they would be useful for a kernel process entered through a means other than a fork interrupt.

A kernel process's stack occupies one or more pages of system space allocated for that purpose when the kernel process is created. The stack has a no-access guard page at each end so that stack underflow and overflow can be detected immediately. Creating a kernel process's stack from one or more pages of virtual memory rather than from nonpaged pool simplifies guard page creation. A typical kernel process's stack is one page long, not counting the guard pages.

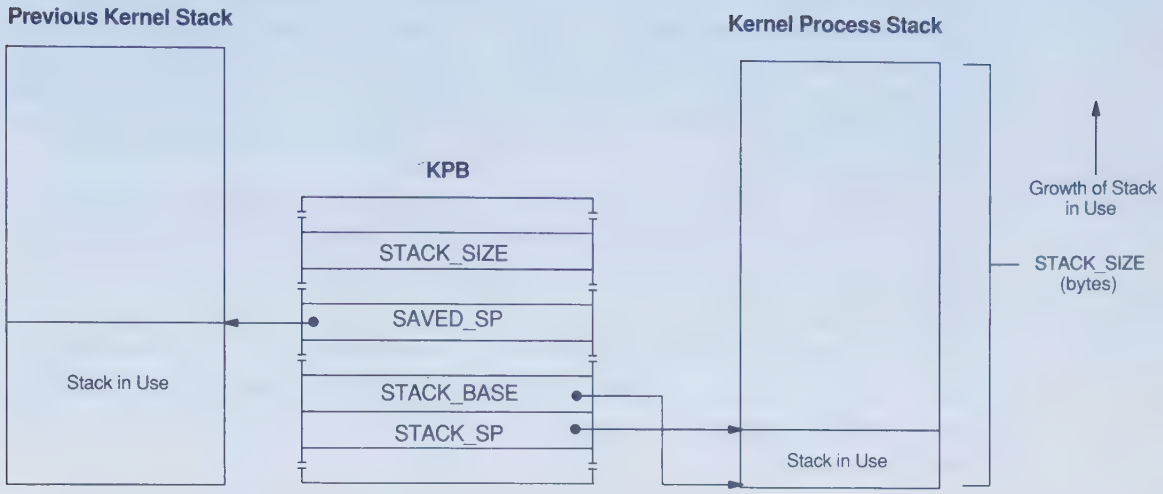
Figure 5–5 shows the stack and the fields in the KPB related to it.

5.5.2 Kernel Process Routines

Routines that support and implement kernel processes include the following:

- EXE$KP_ALLOCATE_KPB and EXE$KP_DEALLOCATE_KPB, to allocate and deallocate the KPB and stack
- EXE$KP_START, to initiate a kernel process
- EXE$KP_STALL_GENERAL, to stall a kernel process
- EXE$KP_RESTART, to restart a kernel process

Figure 5–5 Kernel Process Stack



- EXE$KP_END, to end a kernel process

The sections that follow describe them in detail.

In addition, VMS provides alternative versions of several routines that enable a kernel process to take typical driver fork process actions, such as fork, I/O fork, wait for interrupt, and request channel. Section 5.5.3 describes the routine that implements fork. The others are described in Chapter *I/O Processing*.

5.5.2.1 Routines to Create and Delete Data Structures

EXE$KP_ALLOCATE_KPB, in module KERNEL_PROCESS, is called with arguments specifying the size of the stack, which KPB areas to include, and whether to deallocate the KPB when the kernel process ends. Its most frequent caller is routine EXE$KP_STARTIO, in module KERNEL_PROCESS, a routine used by drivers whose start I/O routines run as kernel processes.

EXE$KP_ALLOCATE_KPB takes the following steps:

1. After validating its argument count, it calculates how big the KPB is to be.
2. It allocates and zeros nonpaged pool for the KPB.
3. It initializes the KPB header, the pointers to its areas, and KPB$IS_FLAGS.
4. To calculate the stack size, EXE$KP_ALLOCATE_KPB rounds up the requested size to the next integral number of pages, based on the processor's page size, and stores the corresponding number of bytes in KPB$IS_STACK_SIZE.
5. It allocates enough system page table entries (SPTEs) to map each page of stack, plus two to serve as guard pages. It stores the address of the high end of stack (excluding the guard pages) in KPB$PS_STACK_BASE.
6. EXE$KP_ALLOCATE_KPB acquires the MMG spinlock, raising IPL to IPL$_MMG, and allocates enough physical pages for the stack.

If it can allocate the stack page or pages, it initializes the page frame number (PFN) database array elements for each page to show the page as an active system page with a reference count of 1. (Chapter *Memory Management Data Structures* describes the PFN database.) It initializes each SPTE that maps a physical page of stack with a protection of KW, valid and address space match (ASM) bits set, and owner of kernel mode, and inserts the PFN. It clears the guard pages' SPTEs.

If it is unable to allocate enough physical pages, it deallocates any allocated, reinitializes any related memory management data structures, and deallocates the KPB to nonpaged pool. It returns the error status SS$_INSFRPGS to its caller.

7. It returns the status SS$_NORMAL and the address of the KPB to its caller.

EXE$KP_DEALLOCATE_KPB, in module KERNEL_PROCESS, is called with the address of a KPB. It takes the following steps to delete the kernel process data structures:

1. After validating its argument count and checking the structure type of the KPB, it sets the KPB$V_DELETING bit in KPB$IS_FLAGS.
2. It initializes the fork block in the KPB to describe an IPL 6 fork process that is to execute the routine CLEANUP_KPB, in module KERNEL_PROCESS.
3. It returns to its caller.

Sometime later, the fork interrupt is granted, and CLEANUP_KPB executes. It takes the following steps:

1. It calculates the address of the SPTE that maps the first (and possibly only) page of the kernel process's stack.
2. It acquires the MMG spinlock, raising IPL to IPL$_MMG.
3. CLEANUP_KPB deallocates the physical page mapped by the SPTE and clears its PFN database array elements. It clears the SPTE and invokes TBI_SINGLE to invalidate any cached translation buffer entry for that page.

   It repeats these actions for any additional stack pages.
4. It deallocates the SPTEs that mapped stack pages and the guard pages' SPTEs and releases the MMG spinlock.
5. It deallocates the KPB to nonpaged pool.
6. CLEANUP_KPB returns to the fork dispatcher.

5.5.2.2 Routines to Enter and Leave a Kernel Process

EXE$KP_START, in module KERNEL_PROCESS_MAGIC, is called with the address of the KPB, the procedure value of the target kernel process routine, and optionally a register save mask. Initially it runs on the stack of its invoker but then switches to the kernel process stack.

EXE$KP_START takes the following steps to initiate a kernel process:

1. It validates its argument count.

2. If a register save mask argument was supplied, EXE$KP_START confirms that it contains no scratch registers (R0, R1, R16–R25, R27, R28, R30, R31) and merges in those that the kernel process routines need saved (R12–R15, R26, R29).

   If one was not supplied, EXE$KP_START stores a default register save mask designating the minimum required registers.

3. It aligns the stack to a quadword boundary so that it can do quadword stores without incurring unaligned data traps. It saves the previous contents of the SP register (that is, the original unaligned stack pointer) and then pushes onto the stack each of the registers specified by the mask, from low register number to high. It saves the value of the updated SP in KPB$PS_SAVED_SP. As a result, R26, the return address in the routine that called EXE$KP_START, is now saved on the current kernel stack.

4. EXE$KP_START loads SP from KPB$PS_STACK_BASE, switching to the kernel process's stack. It initializes KPB$PS_STACK_SP to the contents of KPB$PS_STACK_BASE to reflect a currently empty stack.

5. It sets the active and valid bits in KPB$IS_FLAGS, indicating that the KPB is valid and describes an active kernel process.

6. It pushes onto the stack the return address from the call to EXE$KP_START, its own linkage section pointer, and the address of the KPB. These are on the stack to ensure that when the kernel process returns to EXEKP_START, EXEKP_START has enough information to call EXE$KP_END to end the kernel process. (Typically, control should return to EXE$KP_START's caller when the fork process stalls and the stall routine switches back to the caller's stack and returns.)

7. It calls the target kernel process routine with one argument, the address of the KPB.

EXE$KP_STALL_GENERAL, in module KERNEL_PROCESS_MAGIC, is called from a routine running on that kernel process's stack. It performs the register context save, stack switch, and register context restore necessary to stall a kernel process for any reason. It has one argument, the address of the KPB.

Typically, EXE$KP_STALL_GENERAL is called by a more specific stall routine, which first stores the procedure value of a callback routine in KPB$PS_SCH_STALL_RTN. For example, IOC$KP_WFIKPCH, in module KERNEL_PROCESS, which stalls a fork process to wait for an interrupt or I/O timeout, stores the procedure value of the stall routine STALL_WFIKPCH and then calls EXE$KP_STALL_GENERAL. After performing the steps just summarized, EXE$KP_STALL_GENERAL calls STALL_WFIKPCH, which performs a standard WFIKPCH. Chapter *I/O Processing* details the sequence by which a fork process waits for an interrupt and is reentered when the interrupt occurs or after the wait time has elapsed.

EXE$KP_STALL_GENERAL takes the following steps to stall a kernel process:

1. It validates its argument count.

2. It aligns the stack to a quadword boundary so that it can do quadword stores without incurring unaligned data traps. It saves the previous contents of the SP register (that is, the original unaligned stack pointer) and then pushes each of the registers specified by the mask, from low register number to high. It saves the value of the updated SP in KPB$PS_STACK_SP. As a result, R26, the return address in the routine that called EXE$KP_STALL_GENERAL, is now saved on the current kernel stack.
3. EXE$KP_STALL_GENERAL loads SP from KPB$PS_SAVED_SP, switching to the stack current when the kernel process was entered.
4. It restores the register context saved on that stack at the time of the stack switch onto the kernel process's stack, including the return address in the routine that caused entry to the kernel process and the original unaligned stack pointer.
5. It clears the active bit in KPB$IS_FLAGS, leaving the valid bit set, to indicate that the KPB is valid and describes a stalled kernel process.
6. Passing two arguments, EXE$KP_STALL_GENERAL calls the scheduling stall routine whose procedure value is in KPB$PS_SCH_STALL_RTN. Such a routine performs only those actions that require the kernel process's registers and state to have been saved. The existing scheduling stall routines are driver-related; each inserts a fork block on a queue specific to its stall. STALL_FORK, in module KERNEL_PROCESS, for example, inserts the fork block on a fork queue, from which it will be removed eventually by the fork dispatching interrupt service routine associated with that queue. A scheduling stall routine is called with the address of the KPB and the contents, which can be zero, of KPB$PS_SPL_STALL_RTN. Thus, a scheduling stall routine can call the specified spinlock stall routine, if any.

   The scheduling stall routine saves R3 and R4 in the fork block. It stores in FKB$L_FPC a procedure value within the scheduling stall routine. It queues the fork block and returns to EXE$KP_STALL_GENERAL.
7. When control returns to EXE$KP_STALL_GENERAL, it returns to the routine whose return address was saved on the original kernel stack, the one that caused the most recent entry to the kernel process.

When the fork process is reentered sometime later, the scheduling stall routine will call EXE$KP_RESTART.

EXE$KP_RESTART, in module KERNEL_PROCESS_MAGIC, is called from a routine running on a stack other than that of the kernel process, typically from the restarted fork process. Its function is to restore the kernel process context and stack. It is called with the address of the KPB and optionally a status to return to the kernel process routine being restarted.

EXE$KP_RESTART takes the following steps to restart a kernel process:

1. It validates its argument count.
2. It aligns the stack to a quadword boundary so that it can do quadword stores without incurring unaligned data traps. It saves the previous contents of the SP register (that is, the original unaligned stack pointer) and then pushes each of the registers specified by the mask, from low register number to high.

It saves the value of the updated SP in KPB$PS_SAVED_SP. As a result, R26, the return address in the routine that called EXE$KP_RESTART, is now saved on the current kernel stack.

3. EXE$KP_RESTART loads SP from KPB$PS_STACK_SP, switching to the kernel process's stack.
4. It restores the register context saved on that stack at the time of the stack switch from the kernel process's stack, including the return address in the routine that stalled the kernel process and the original unaligned stack pointer.
5. It sets the active bit in KPB$IS_FLAGS to indicate that the KPB describes an active kernel process.
6. EXE$KP_RESTART tests the field KPB$PS_SCH_RESTRT_RTN and, if it is zero, continues with the next step. Currently, all kernel process stall routines clear this KPB field.

   If the field is nonzero, EXE$KP_RESTART calls the routine whose procedure value is in KPB$PS_SCH_RESTRT_RTN, passing it three arguments: the address of the KPB, the status to be passed back to the kernel process, and the contents of KPB$PS_SPL_RESTRT_RTN. Thus, a scheduling restart routine can call the specified spinlock restart routine, if any.
7. EXE$KP_RESTART returns to the kernel process routine that had stalled, the one that called EXE$KP_STALL_GENERAL.

EXE$KP_END, in module KERNEL_PROCESS_MAGIC, is called with the address of the KPB from a routine running on that kernel process's stack. Its function is to terminate the kernel process and return to the routine that most recently caused entry into the kernel process.

EXE$KP_END takes the following steps to terminate a kernel process:

1. After validating its argument count, it switches back to the stack whose pointer was saved in KPB$PS_SAVED_SP and zeros that field to prevent its reuse.
2. It restores the context saved on that stack, namely, the registers specified by KPB$IS_REG_MASK, and then restores the original value of the stack pointer, popping off any alignment bytes.
3. It clears the active and valid bits in KPB$IS_FLAGS.
4. If KPB$V_DEALLOC_AT_END is set, its typical state, EXE$KP_END calls EXE$KP_DEALLOCATE_KPB, which deallocates the KPB and kernel process stack.
5. It then returns to the routine that had most recently caused entry to the kernel process. Its return address was restored to R26 in step 2.

5.5.3 Example Kernel Process Flow

Figure 5–6 shows a partial possible flow of a kernel process that runs as a fork process. Some kernel mode thread of execution is running and enters the kernel process, perhaps by calling EXE$KP_START. EXE$KP_START switches to the kernel process's stack.

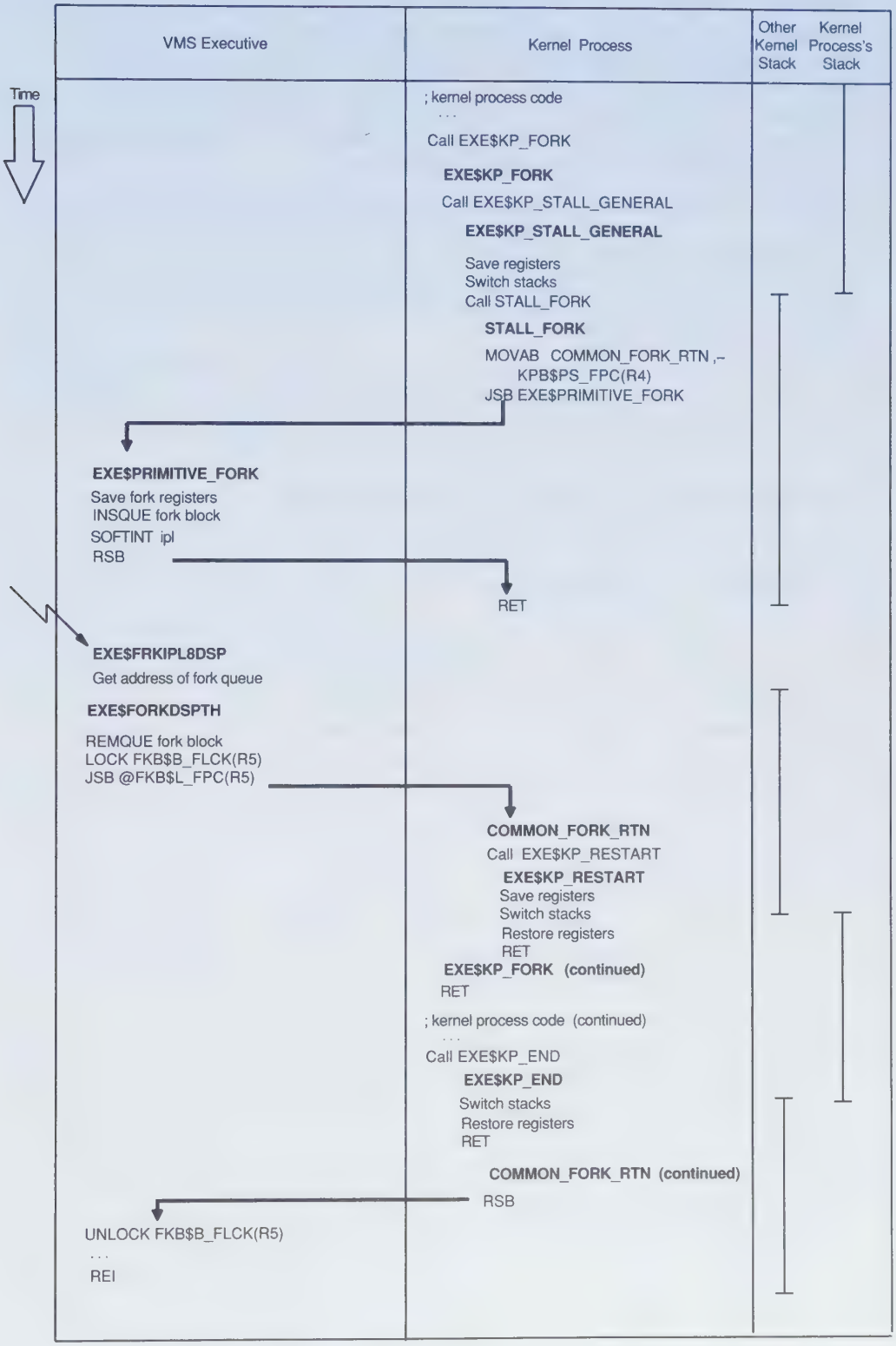
The figure shows the interaction between the kernel process and the standard fork routines as well as the flow among some of the kernel process routines previously described. The rightmost column in the figure shows the stack switches. Each vertical bar in the column represents stack execution: the right-hand ones, execution on the kernel process's stack, and the left-hand ones, execution on another kernel stack.

After the kernel process has been entered, the flow proceeds as follows:

1. The kernel process calls EXE$KP_FORK, in module KERNEL_PROCESS, to fork. EXE$KP_FORK is called with the address of a KPB and optionally the address of a fork block.
2. EXE$KP_FORK takes the following steps:
   a. EXE$KP_FORK determines whether the second argument was passed to it and, if not, gets the address of the fork block in the KPB.
   b. It stores the procedure value of its routine STALL_FORK in KPB$PS_SCH_STALL_RTN and clears KPB$PS_RESTRT_RTN.
   c. It calls EXE$KP_STALL_GENERAL to stall the kernel process using the routine STALL_FORK.

   Control will return to EXE$KP_FORK when the requested fork process executes.
3. EXE$KP_STALL_GENERAL saves the register context of the kernel process, including the return address of EXE$KP_FORK, and switches stacks. EXE$KP_STALL_GENERAL calls STALL_FORK.
4. STALL_FORK stores the procedure value of the local routine COMMON_FORK_RTN in FKB$L_FPC. It invokes the FORK macro, described in Section 5.4.3, which transfers control to EXE$PRIMITIVE_FORK.
5. EXE$PRIMITIVE_FORK, in module FORKCNTRL, saves the fork registers, inserts the KPB's fork block on the fork queue corresponding to its spinlock's IPL, and requests a software interrupt at that IPL (in this example, the IPL 8 queue). It returns to STALL_FORK.
6. STALL_FORK returns to EXE$KP_STALL_GENERAL, which returns to the routine that entered the kernel process, that is, the routine that called EXE$KP_START or EXE$KP_RESTART. (The latter part of this step is not shown in the figure.)
7. Sometime later, the fork interrupt is granted. The interrupt service routine executes on whichever kernel stack is current. It could be the kernel stack of any process, the system process's stack, or a different kernel process's stack. EXE$FORKDSPTH removes the KPB's fork block from the queue, acquires

Figure 5–6 Example Kernel Process Flow



the specified spinlock, and dispatches to the routine whose procedure value is stored in FKB$L_FPC, COMMON_FORK_RTN.

8. COMMON_FORK_RTN calls EXE$KP_RESTART. Control will return to COMMON_FORK_RTN when the kernel process stalls again or ends.
9. EXE$KP_RESTART saves the register context, switches stacks to that of the kernel process, and restores its register context. It returns to EXE$KP_STALL_GENERAL's caller, EXE$KP_FORK.
10. EXE$KP_FORK returns to its caller, the kernel process routine.
11. If the kernel process has completed its task, it can either call EXE$KP_END itself or return to EXE$KP_START, which will call EXE$KP_END. If the kernel process has not completed its task but has to wait before proceeding further, it can stall again.
12. Whether called by the kernel process or by EXEKP_START, EXEKP_END switches stacks to the stack on which the fork interrupt occurred and optionally deallocates the KPB and kernel stack. It returns to the routine that entered the kernel process, COMMON_FORK_RTN.
13. COMMON_FORK_RTN returns to the fork dispatcher.

5.6 Revision Information

Source Module	Build Date
[SYS]IPCONTROL.MAR	31-May-1991
[LIB]FKBDEF.SDL	12-June-1991
[LIB]KPBDEF.SDL	12-June-1991
[SYS]FORKCNTRL.MAR	12-June-1991
[SYS]KERNEL_PROCESS.MAR	12-June-1991
[SYS]KERNEL_PROCESS_MAGIC.M64	12-June-1991
[SYS]SYSTEM_ROUTINES.M64	13-August-1991

Chapter 7

System Service Dispatching

Between the idea
And the reality
Between the motion
And the act
Falls the Shadow.

T. S. Eliot, *The Hollow Men*

Many of the operations that the VMS operating system performs on behalf of a user are implemented as procedures called system services. A user application requests system services directly, and components such as the file system request system services on behalf of the user.

Most system service procedures are contained in executive images and reside in system space; others are contained in privileged shareable images.

System services typically execute in kernel or executive access mode so that they can read and write data structures protected from access by outer modes.

This chapter describes how control is passed from an image to a procedure that executes service-specific code and back to the image.

7.1 Overview

A system service is a procedure in the VMS executive that performs a function for a process, usually at the process's request. An image running in the process requests a particular service by calling the associated procedure.

A major distinction between system service procedures and other procedures is that system services are generally provided by the executive. In early versions of VAX/VMS, system services were provided by system service procedures in SYS.EXE, the system image, and Record Management Services (RMS) were provided by procedures in RMS.EXE. Since then, the system service mechanism has been extended to user-written services in privileged shareable images, and the system image has been reorganized into executive images, eliminating most distinctions between system and RMS services. Section 7.2 describes system services in executive images, and Section 7.7, system services in privileged shareable images.

Another important distinction between a system service procedure and, for example, a Run-Time Library procedure, is access mode. Most system service procedures execute in kernel or executive mode and may be requested from any mode equally or less privileged. (Some system services, however, known as mode of caller services, execute in the same mode as their requestor and can be requested from any mode.) Run-Time and other library procedures generally execute in user mode and may only be requested from user mode.

The implementation of inner mode system services is based on a controlled change of access mode. Although an unprivileged process can enter an inner access mode to execute code in that mode, it can only execute procedures that are part of the executive or that have been specifically installed by the system manager. Only a process with suitable privilege may change to an inner access mode and execute a procedure of its own designation (see Section 7.8.2).

When a process requests an inner mode system service, it executes a system service transfer routine that serves as a bridge between the requestor's mode and the inner mode in which the service procedure executes. A system service transfer routine contains an instruction that loads R0 with a number identifying the system service and either a `CALL_PAL CHMK` or a `CALL_PAL CHME` instruction. Executing one of these instructions causes a CHMK or CHME exception; the CHMK and CHME exception service routines are called change mode dispatchers. A change mode dispatcher transfers control to a procedure that is identified by the number in R0 and that actually implements the service. Section 7.2.3 describes system service transfer routines; Section 7.4, change mode instructions; and Sections 7.5.2 and 7.5.3, the operations of the change mode dispatchers.

A key goal for the implementation of VAX/VMS system services has been that an image requesting system services need not be relinked from one version of VMS to the next. This constraint was initially met by forever reserving the lowest pages of system space for system service transfer routines (called system service vectors under VAX/VMS). Each system service transfer routine was given a global name whose value was an address within the reserved pages. In VAX/VMS Version 3 and subsequent versions, a region of P1 space was also reserved for this purpose.

The [*censored*] VMS implementation of system services has the same goal: that an image requesting a system service need not be relinked from one version of the system to the next. However, [*censored*] VMS satisfies the goal differently. System service transfer routine names are resolved using the same mechanisms that apply to externally visible names in a shareable image. Section 7.2.1 briefly summarizes the concept of a shareable image symbol vector and explains its use in resolving system service transfer routine names.

Section 7.8 describes several system services related to change mode dispatching.

7.2 Transfer Routines for System Services in Executive Images

This section discusses transfer routines for system services in VMS executive images: the resolution of their externally visible names, their initialization, and their contents. Section 7.7 contains similar information about transfer routines for system services in privileged shareable images.

The external name of a system service transfer routine generally has the form SYS$*service*. A transfer routine executes in the mode of the caller and serves as a bridge between the caller and the procedure that actually implements the service request. That procedure is typically part of an executive image and typically executes in an inner access mode. The name of a procedure that performs the actual work of the system service is usually of the form EXE$*service* or RMS$*service*.

7.2.1 Resolving System Service Transfer Routine Names

Every [*censored*] procedure is described by a data structure called a procedure descriptor that contains the address of the procedure's code entry point and information about its type and characteristics. (The term *procedure value* refers to the address of a procedure descriptor.) A compiler generates a procedure descriptor for each procedure in a module and places them together in a program section called a linkage section.

Executive image system service transfer routines and their associated procedure descriptors are defined in the module SYSTEM_SERVICES.

The compiled representation of a call from code in one module to a procedure in another module (an external procedure) typically includes a two-quadword data structure called a linkage pair. A compiler generates linkage pairs in a linkage section. By the time a call using a linkage pair is executed, the first quadword of the linkage pair must contain the external procedure's code entry point address, and the second quadword must contain the address of its procedure descriptor (its procedure value). When the linker links an object module containing a call with the object module containing the called procedure, the linker fills in the linkage pair in the calling object module.

When the linker links an executable image containing a call to an external procedure in a shareable image, the linker cannot resolve the contents of the linkage pair because a shareable image is not assigned address space until it is activated. Instead, the linker records in the image the need for the image activator to resolve the contents. Chapter *Image Activation and Exit* describes image activation, fixup, and relocation.

The global symbols from object modules in a shareable image that are to be visible externally are called universal symbols. All universal symbols must be identified to the linker in the options with which the shareable image is linked. The linker creates a universal symbol for each global symbol identified in the SYMBOL_VECTOR option. To describe the universal symbols, the linker creates within the shareable image both an image section called a symbol vector and a global symbol table.

A symbol vector contains two quadwords for each universal symbol. For a universal symbol that is the name of a procedure, the two quadwords will hold the addresses of the procedure's code entry point and its procedure descriptor. That is, they form replacement contents for a linkage pair. A global symbol table lists the shareable image's universal symbols and their values. The value of a universal symbol, as recorded in a global symbol table, is the offset of its two quadwords from the beginning of the symbol vector. (In contrast, the value of a VAX/VMS universal symbol is its offset in the shareable image.)

During image activation, address space is assigned to a shareable image, and its symbol vector entries are updated to contain actual addresses of the procedure descriptors and code entry points. Subsequently, in any image that is activated with a shareable image and that contains a call to it, the image activator resolves the linkage pair for the call by replacing the linkage pair's contents with the corresponding replacement contents from the symbol vector of the shareable image.

Calls to system services are resolved in the same way. Specifically, the module SYSTEM_SERVICES is linked to create the shareable image SYS$PUBLIC_VECTORS.EXE. Its global symbol table lists all the system service transfer routine names and their offsets in the image's symbol vector.

When an image that requests a system service is linked, the linker must resolve the system service transfer routine name. By default, it searches the file SYS$LIBRARY:SYS$PUBLIC_VECTORS.EXE to resolve any still unresolved symbols after it searches STARLET.OLB. The linker stores the symbol vector offset corresponding to the system service transfer routine into the linkage section of the image making the service request. It stores information about the need to update that linkage pair in the fixup section of the image.

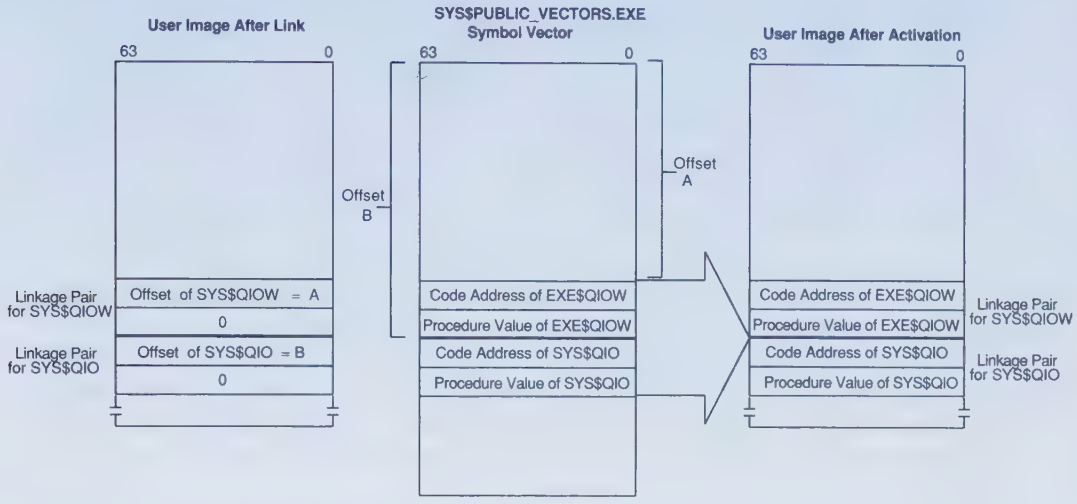
Unlike other shareable images, SYS$PUBLIC_VECTORS.EXE is not activated with an executable image. Instead, as part of the executive, it is loaded into system space during system initialization. Because all processes map system space, SYS$PUBLIC_VECTORS.EXE is shared. The addresses in its symbol vector are updated as system services are loaded (see Section 7.2.2 and Chapter *The Modular Executive*).

When an image that requests a system service is activated, the image activator sees that the image has been linked with SYS$PUBLIC_VECTORS.EXE, which it treats as a shareable image that has already been activated. The image activator replaces the contents of the linkage pair representing the system service transfer routine with the two quadwords of information in the loaded SYS$PUBLIC_VECTORS.EXE symbol vector.

Figure 7–1 shows how references to system services are resolved in a linked user image and how they are fixed up after the image is activated. After the image is linked, its linkage pairs that refer to system services contain the offsets of those symbols in the SYS$PUBLIC_VECTORS.EXE symbol vector. For simplicity, the figure places the symbol vector entries for SYS$QIO and SYS$QIOW in adjacent positions and omits the user image's fixup section, which contains a linkage pair type of fixup for each of these references. Section 7.2.2 explains the difference between the symbol vector entries for SYS$QIO (an inner mode service) and SYS$QIOW (a mode of caller service). When the image is activated, the linkage

pairs are fixed up to contain the replacement linkage pair contents at those offsets in the loaded SYS$PUBLIC_VECTORS.EXE symbol vector.

Figure 7–1 System Service Name Resolution



The order of the entries in a symbol vector is determined by the order in which global symbols are listed in the SYMBOL_VECTOR link option. To ensure upward compatibility of images linked with a shareable image, the SYMBOL_VECTOR link option symbols must remain in the same order from one version of the shareable image to the next. New symbols may be added at the end, and a deleted symbol must be replaced by a new symbol or the name of a placeholder procedure. To ensure that a *censored* VMS image that references system services can run without relinking on a new version of *censored* VMS, VMS guarantees the order of the symbols defined in SYS$PUBLIC_VECTORS.EXE.

7.2.2 Initialization of System Service Transfer Routines and Procedure Descriptors

SYS$PUBLIC_VECTORS.EXE contains not only the symbol vector for system service transfer routine names but also their procedure descriptors and the system service transfer routines themselves. During system initialization, SYS$PUBLIC_VECTORS.EXE is loaded into system space. The base image cells EXE$GL_PUBLIC_VECTOR_SYMVEC and EXE$GL_PUBLIC_VECTOR_SYMVEC_END contain the beginning and end addresses of the symbol vector within SYS$PUBLIC_VECTORS.EXE.

The procedure descriptor for a system service transfer routine is a particular type called a bound procedure descriptor. A bound procedure descriptor contains the address of a second procedure descriptor to which the first is bound. This provides a mechanism by which the target of a procedure call can be resolved at run time.

Each system service transfer routine procedure descriptor contains the code entry point of a transfer routine in module SYSTEM_SERVICES. At assembly time, each transfer routine is defined as a call to the procedure to which its procedure descriptor is currently bound. Initially the procedure descriptor is bound to SYS$LOAD_ERROR, in module SYSTEM_SERVICES. SYS$LOAD_ERROR simply returns the error status SS$_ILLSER to the system service requestor.

An executive image module that contains system services procedures contains one invocation of the SYSTEM_SERVICE macro for each of them. This macro labels the procedure and creates a system service descriptor block that describes the system service: the address of its own system service transfer routine, access mode, and other characteristics (see Chapter *The Modular Executive*).

When an executive image containing one or more services is loaded, the routine EXE$CONNECT_SERVICES, in module SYSTEM_SERVICE_LOADER, uses the system service descriptor block to determine the modifications to be made to the system service transfer routine and its procedure descriptor, both in the loaded SYS$PUBLIC_VECTORS.EXE. For an inner mode service, it leaves the procedure descriptor alone, assigns a change mode number, and replaces the transfer routine's procedure call with a `CALL_PAL CHMx`. Section 7.2.3 shows the resulting transfer routine.

For a mode of caller service, EXE$CONNECT_SERVICES leaves the transfer routine alone and simply binds the transfer routine's procedure descriptor in SYS$PUBLIC_VECTORS.EXE to the newly loaded procedure in the executive image. More significantly, EXE$CONNECT_SERVICES modifies the system service's linkage pair replacement contents in the symbol vector. When an image is activated that requests an already loaded mode of caller service, its linkage pair is fixed up with the addresses of the procedure descriptor and code entry point of the procedure in the executive image rather than those of the transfer routine. If an image loaded prior to the service connection calls the service, it will execute the transfer routine, which calls the real service procedure instead of SYS$LOAD_ERROR.

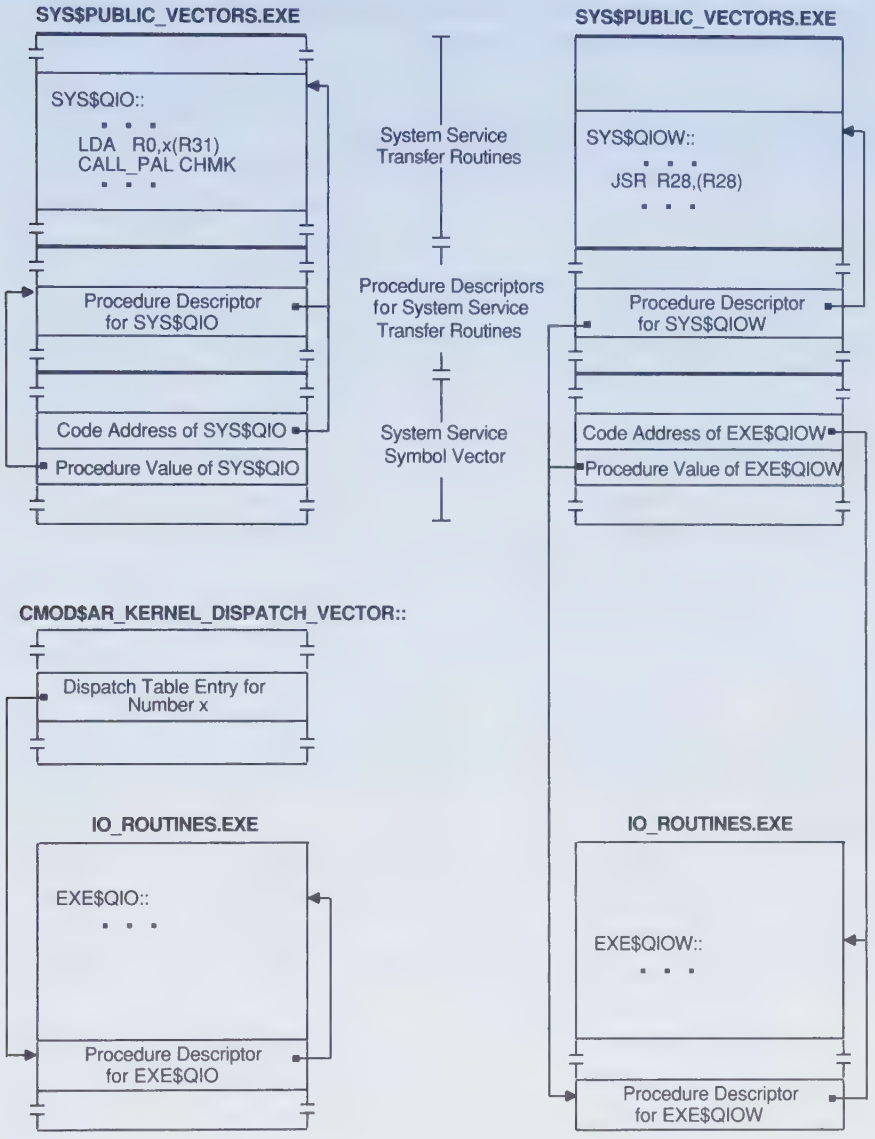
Figure 7–2 shows the relations among system service transfer routines, their procedure descriptors, and the system service symbol vector after system services are connected. It contrasts the structures and routine for the kernel mode service Queue I/O Request ($QIO), on the left side of the picture, with those for the mode of caller service Queue I/O Request and Wait ($QIOW), on the right side of the picture.

Chapter *The Modular Executive* contains more details about system service initialization.

7.2.3 System Service Transfer Routines

A system service transfer routine is generally identified by a universal name of the form SYS$*service*. The routine consists of instructions that cause a transfer of control to a service-specific procedure in the executive.

Figure 7–2 System Service Transfer Routines



Many system services execute in kernel mode; after the services are loaded, their system service transfer routines contain a CALL_PAL CHMK instruction. The transfer routines for a few system services and many RMS services contain a CALL_PAL CHME instruction. Some services, such as the text-formatting services, execute in the access mode of the caller. When an image calls such a service, it dispatches directly to the service-specific procedure in the executive that implements the service.

Following are the three sets of instructions found in system service transfer routines. Table 7–1 lists the VMS system services that use each of these three methods of initial dispatch.

Table 7–1 System Services and RMS Services That Use Each Form of System Service Transfer Routine

The following services execute initially in kernel mode.		
$ADJSTK	$DEVICE_SCAN	$RESUME
$ADJWSL	$DGBLSC	SAVE_SYS_ALIGN_FAULT_REPORT
$ALLOC	$DIAGNOSE	$SCHDWK
$ASCEFC	$DLCEFC	$SETEF
$ASSIGN_LOCAL	$ENQ	$SETEXV
$BRKTHRU	$ERAPAT	$SETFLT
$CANCEL	$EXIT	$SETIME
$CANEXH	$EXPREG	$SETIMR
$CANTIM	$FORCEX	$SETPFM
$CANWAK	GET_SYS_ALIGN_FAULT_REPORT	$SETPRA
$CHKPRO	$GETCHN[1]	$SETPRI
$CLRAST	$GETDEV[1]	$SETPRN
$CLRAST_MODE	$GETDVI	$SETPRT
$CLREF	$GETJPI	$SETPRV
$CMKRNL	$GETLKI	$SETRWM
$CNTREG	$GETPTI	$SETSFM
$CREATE_BUFOBJ	$GETSECI	$SETSSF
$CRELNM	$GETSYI	$SETSTK
$CRELNT	$GETTIM	$SETSWM
$CREMBX	$HIBER	$SIGPRC
$CREPRC	INIT_SYS_ALIGN_FAULT_REPORT	$SNDERR
$CRETVA	$KRNDWN	$SUBSYSTEM
$CRMPSC	$LCKPAG	$SUSPND
$DACEFC	$LKWSET	$TRNLNM
$DALLOC	$MGBLSC	$ULKPAG
$DASSGN	$MTACCESS	$ULWSET
$DCLAST	$PROCESS_SCAN	$UPDSEC
$DCLCMH	PTD$CANCEL[2]	$WAITFR_COMMON
$DCLEXH	PTD$CREATE[2]	$WAITFR_INT
$DELETE_BUFOBJ	PTD$DECTERM_SET_PAGE_SIZE[2]	$WAKE
$DELLNM	PTD$DELETE[2]	$WFLAND_COMMON
$DELMBX	PTD$READ[2]	$WFLAND_INT

[1]This service has been superseded.

[2]This service is implemented in a privileged shareable image.

Table 7–1 (Cont.) System Services and RMS Services That Use Each Form of System Service Transfer Routine

The following services execute initially in kernel mode.		
$DELPRC	PTD$SET_EVENT_NOTIFICATION[2]	$WFLOR_COMMON
$DELTVA	PTD$WRITE[2]	$WFLOR_INT
$DEQ	$PURGWS	
The following system services execute initially in executive mode.		
$ADD_HOLDER[2]	$FIND_HELD[2]	$MOD_IDENT[2]
$ADD_IDENT[2]	$FIND_HOLDER[2]	$NUMTIM
$ASCTOID	$FINISH_RDB	$REM_HOLDER[2]
$CHANGE_ACL[2]	$FORGE_WORD[2]	$REM_IDENT[2]
$CHANGE_CLASS[2]	$GETQUI	$SETUAI[2]
$CHECK_ACCESS[2]	$GETUAI[2]	$SNDACC[1]
$CREATE_RDB[2]	$IDTOASC	$SNDJBC
$DISMOU[2]	$IMGACT	$SNDOPR
$ERNDWN	$MOD_HOLDER[2]	$SNDSMB[1]
The following system services execute initially in the mode of the caller. Several of them change to a more privileged mode during their execution.		
$ASCTIM	$GETMSG[4]	$REVOKID[4]
$ASSIGN	$GRANTID[4]	$RUNDWN
$BINTIM	$HASH_PASSWORD	$SETAST
$BRDCST[1]	$IMGFIX	START_ALIGN_FAULT_REPORT
$CHECK_FEN	$IMGSTA[3]	STOP_ALIGN_FAULT_REPORT
$CRELOG[1]	$INIT_VOL[2,4]	$TRNLOG
$DELLOG[1]	$MOUNT[2,4]	$UNWIND
$EXCMSG[4]	$PARSE_ACL[2]	$WAITFR
$FAO	$PARSE_CLASS[2]	$WFLAND
$FAOL	PERM_DIS_ALIGN_FAULT	$WFLOR
$FORMAT_ACL[2]	PERM_REPORT_ALIGN_FAULT	$WRITE_THREAD_UNQ
$FORMAT_CLASS[2]	$PUTMSG[3]	

[1]This service has been superseded.

[2]This service is implemented in a privileged shareable image.

[3]This system service can be called only from supervisor and user modes.

[4]This system service can be called only from executive and less privileged access modes.

Table 7–1 (Cont.) System Services and RMS Services That Use Each Form of System Service Transfer Routine

The following system services execute initially in the mode of the caller. Several of them change to a more privileged mode during their execution.		
GET_ALIGN_FAULT_DATA	$READ_THREAD_UNQ	
The following mode of caller system services are synchronous services. Each requests the asynchronous form of the service and then a wait-for service.		
$BRKTHRUW	$GETQUIW	$SNDJBCW
$ENQW	$GETSYIW	$SYNCH
$GETDVIW	PTD$READW[2]	$UPDSECW
$GETLKIW	$QIOW	
The following RMS services execute in executive mode.		
$FILESCAN	$RMS_FREE	$RMS_SEARCH
$RMS_CLOSE	$RMS_GET	$RMS_SPACE
$RMS_CONNECT	$RMS_MODIFY	$RMS_TRUNCATE
$RMS_CREATE	$RMS_NXTVOL	$RMS_UPDATE
$RMS_DELETE	$RMS_OPEN	$RMS_WAIT
$RMS_DISCONNECT	$RMS_PARSE	$RMS_WRITE
$RMS_DISPLAY	$RMS_PUT	$RMSRUNDWN
$RMS_ENTER	$RMS_READ	$SETDDIR
$RMS_ERASE	$RMS_RELEASE	$SETDFPROT
$RMS_EXTEND	$RMS_REMOVE	$SSVEXC
$RMS_FIND	$RMS_RENAME	
$RMS_FLUSH	$RMS_REWIND	
The following RMS services are mode of caller services. Each requests the executive mode form of the service and then optionally waits.		
$CLOSE	$FLUSH	$REMOVE
$CONNECT	$FREE	$RENAME
$CREATE	$GET	$REWIND
$DELETE	$MODIFY	$SEARCH
$DISCONNECT	$NXTVOL	$SPACE
$DISPLAY	$OPEN	$TRUNCATE
$ENTER	$PARSE	$UPDATE
$ERASE	$PUT	$WAIT
$EXTEND	$READ	$WRITE
$FIND	$RELEASE	

[2]This service is implemented in a privileged shareable image.

A transfer routine for a system service that changes mode to kernel consists of the following code:

```
SYS$service::                       ;Entry point
        BIS     SP,R31,R28          ;Copy stack pointer
        LDA     R0,chmk_code(R31)
                                    ;Load service-specific value
        CALL_PAL CHMK               ;Change mode to kernel
        RET     R31,(R26)           ;Return to service caller
        NOP                         ;No-op for padding
        NOP                         ;No-op for padding
```

A transfer routine for a system service that changes mode to executive consists of the following code:

```
SYS$service::                       ;Entry point
        BIS     SP,R31,R28          ;Copy stack pointer
        LDA     R0,chme_code(R31)
                                    ;Load service-specific value
        CALL_PAL CHME               ;Change mode to executive
        RET     R31,(R26)           ;Return to service caller
        NOP                         ;No-op for padding
        NOP                         ;No-op for padding
```

A transfer routine for a mode of caller system service consists of the following code:

```
SYS$service::                       ;Entry point
        LDQ     R27,16(R27)         ;Load procedure value
        LDQ     R28,8(R27)          ;Load code entry address
        JSR     R28,(R28)           ;Transfer to procedure
        NOP                         ;No-op for padding
        NOP                         ;No-op for padding
        NOP                         ;No-op for padding
```

An image requesting a mode of caller service that is activated after the service is loaded dispatches directly to the mode of caller service without going through the system service transfer routine. Only service requests from executive images loaded prior to the image with the system service execute its transfer routine.

7.2.4 System Service Requests from Translated Images

A VAX/VMS image that executes on a VAX/VMS system can call a system service through either the P1 or system space address of its vector, although use of system space vector addresses is rare. To be translatable, a VAX/VMS image must restrict itself to the default P1 space resolution of system service vector addresses.

The translated version of a VAX/VMS system service request is a call to a procedure at the same address as the corresponding VAX/VMS system service vector. In [*censored*] VMS, however, the procedure is a jacket routine provided by the translated image environment. The jacket routine makes any necessary argument list transformations and calls the corresponding [*censored*] VMS system service. Chapter *Translated Image Environment* provides more details on support for translated images.

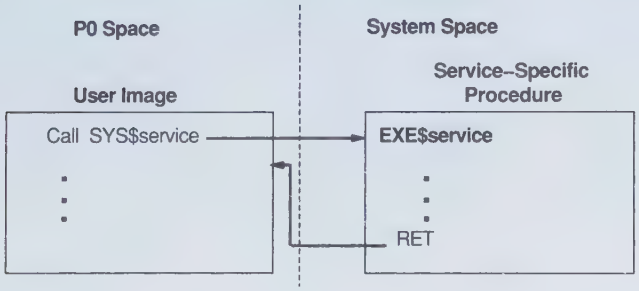
7.3 System Services That Do Not Change Mode

Some system services do not change to a more privileged access mode and instead execute in the mode from which they were requested. The system service transfer routine for one of these mode of caller services transfers control directly to the service procedure. However, as previously described, an image requesting a mode of caller service transfers control directly to the system service-specific procedure (unless it is an executive image loaded prior to the image containing the service-specific procedure).

When the service-specific procedure has completed its operation, it loads a status into R0 and returns control to the requestor of the service at the instruction following the system service request. Table 7–1 lists the mode of caller VMS system services.

Figure 7–3 shows the control flow from a user image to a mode of caller service procedure.

Figure 7–3 Control Flow of System Services That Do Not Change Mode



7.4 Change Mode Instructions

There are four change mode instructions: `CALL_PAL CHMU`, `CALL_PAL CHMS`, `CALL_PAL CHME`, and `CALL_PAL CHMK`. Unlike a VAX `CHMx` instruction, a [*censored*] `CALL_PAL CHMx` instruction does not have an operand to identify the change mode number. Instead, a register must be loaded with a change mode number before the [*censored*] instruction is executed. The VMS change mode dispatchers expect the number to have been loaded into R0 before a `CALL_PAL CHME` or `CALL_PAL CHMK` instruction is executed.

Executing any of the `CALL_PAL CHMx` instructions generates an exception. The privileged architecture library (PALcode) instructions that initiate exceptions and interrupts (referred to in this book as the IEI PALcode routine) alter the access mode, if necessary, and switch to that mode's stack. The actual access mode used is the innermost of the access mode indicated by the instruction and the current access mode. The IEI PALcode pushes onto that stack the processor status (PS), the program counter (PC) of the next instruction, and R7 through R2, forming an

exception frame. It then dispatches through the system control block (SCB) entry for that CHM*x* exception to its exception service routine.

VMS uses the CALL_PAL CHME and CALL_PAL CHMK instructions exclusively to request system services. Their exception service routines are known as the change mode dispatchers.

CHMS and CHMU exceptions are treated much like other exceptions that VMS passes to a user-declared condition handler (see Chapter *Condition Handling*).

7.5 Change Mode Dispatching

Each change mode dispatcher processes the argument list and, using the CHM*x* number, indexes into a dispatch table. From the entry in the dispatch table, it gets the address of the system service procedure descriptor and the code entry address and calls the procedure.

7.5.1 Change Mode Dispatcher Data Structures

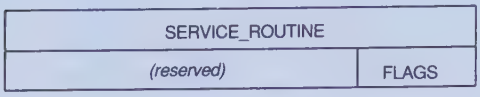
Each change mode dispatcher has its own dispatch table to describe loaded services and a cell representing the number of loaded services. These tables and cells are defined in module CHANGE_MODE_DATA. Each dispatch table has a maximum of 256 entries. The executive mode dispatch table begins at CMOD$AR_EXEC_DISPATCH_VECTOR, and the kernel mode table at CMOD$AR_KERNEL_DISPATCH_VECTOR. The cells containing counts of loaded services are CMOD$GL_CHMK_LIMIT and CMOD$GL_CHME_LIMIT. Each is initialized to zero.

Figure 7–4 shows the format of an individual dispatch table entry. The macro $DISPDEF defines symbolic names for its fields:

- The service routine procedure value field contains the address of the procedure descriptor for the service-specific procedure. The name of a service-specific procedure in an executive image is generally of the form EXE$*service* or RMS$*service*.
- The flags field has two flags, whose symbolic names are defined by the $SSDESCRDEF macro:
  - — SSFLAG_K_WCM, when set, means the service can return the status SS$_WAIT_CALLERS_MODE (see Section 7.5.3).
  - — SSFLAG_K_WCM_NO_REEXEC, when set, means the service can return the status SS$_WAIT_CALLERS_MODE but should not reexecute the service (see Section 7.5.3).

  Use of these flags is reserved to kernel mode services.

Figure 7–4 Change Mode Dispatch Table Entry



When an executive image containing one or more system services is loaded, routine EXE$CONNECT_SERVICES, in module SYSTEM_SERVICE_LOADER, assigns it a unique change mode number equal to the contents of CMOD$GL_CHM*x*_LIMIT. EXE$CONNECT_SERVICES also initializes the dispatch table entry indexed by that change mode number and increments CMOD$GL_CHM*x*_LIMIT. Section 7.2.2 describes modifications it makes to the system service transfer routine or its procedure descriptor. Chapter *The Modular Executive* describes the actual image loading and address space assignment.

7.5.2 Operations of the Change Mode Dispatchers

Module SYSTEM_SERVICE_DISPATCHER contains the change mode dispatchers: EXE$CMODKRNL for CHMK exceptions and EXE$CMODEXEC for CHME exceptions. Their operations are almost identical. The following description applies to both dispatchers, unless otherwise specified:

1. Each dispatcher extracts the access mode from the PS in the exception frame and records it in the current PS's software bits as the previous mode.
2. It compares the contents of R0, the service-specific change mode number, with those of CMOD$GL_CHMK_LIMIT or CMOD$GL_CHME_LIMIT. If the change mode number is equal to or greater than the appropriate limit, the dispatcher checks whether the service is supplied as part of a privileged shareable image, as described in Section 7.7.

   If the change mode number is less than the limit, the service procedure is part of an executive image.
3. Each dispatcher indexes its dispatch table with the change mode number.

   Examining the dispatch table entry, the kernel mode dispatcher, EXE$CMODKRNL, tests whether SSFLAG_K_WCM is set. (This mechanism is only supported for kernel mode services.) If so, the service can return the status SS$_WAIT_CALLERS_MODE with the intent that the service reexecute. If the flag is set, EXE$CMODKRNL copies the argument information (AI) register and the argument registers (R16 through R21) to the kernel stack. Later, EXE$CMODKRNL will be able to restore them from the stack if the service returns a status indicating that the process should be placed into a wait state.

   Thus, the service can be rerequested after the wait with the same arguments even if the service procedure overwrote those registers. (Because no floating-point argument registers are saved, a service that uses this mechanism cannot rely on restored floating-point argument registers and must refrain from overwriting them.)

4. Each dispatcher tests the low byte of the AI register to determine the number of arguments being passed to the system service. If there are more than six, the seventh and subsequent arguments are currently on the previous mode stack. Each dispatcher executes a CALL_PAL PROBER instruction to confirm that the previous mode has read access to the list. If not, it dismisses the change mode exception, returning the error status SS$_ACCVIO to the system service requestor.

   Each dispatcher copies the seventh and subsequent arguments from the outer mode stack to the inner mode stack, where the calling standard requires them to be at entry to the service procedure.

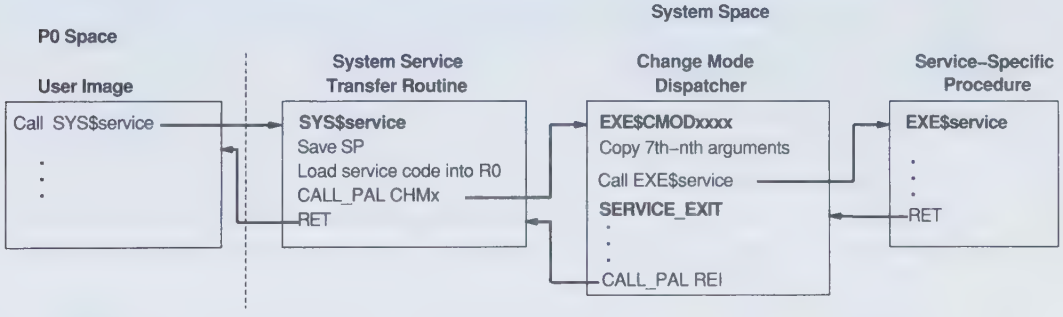
5. EXE$CMODKRNL loads the address of the process's process control block (PCB) into R4.

6. Each dispatcher saves the return address (RA) register contents so that later it can return to the system service requestor.

7. Each dispatcher gets the procedure value of the service-specific procedure from the dispatch table entry associated with the change mode number and calls it.

Figure 7–5 illustrates the control flow from the user program to the service-specific procedure. This flow is the same for both kernel and executive access modes.

Figure 7–5 Control Flow of System Services That Change Mode



The first code in each executive image service-specific procedure, apart from compiler-generated prologue code, is code generated by the SYSTEM_SERVICE macro to check that the service has been called with at least the required number of arguments. If not, the code loads the status SS$_INSFARG into R0 and immediately returns to the change mode dispatcher.

7.5.3 Change Mode Dispatcher Common Exit Path

When a service-specific procedure has completed its operation, it loads a status into R0 and returns to the change mode dispatcher at SERVICE_EXIT.

SERVICE_EXIT takes the following steps:

1. It tests the status returned in R0. If the status is a success or warning, the routine resets the stack pointer, leaving only the exception frame. It executes a CALL_PAL REI instruction, changing access mode to the mode from which the service request was made and transferring control to the RET instruction in the system service transfer routine. Execution of the RET instruction returns control to the address in R26, the instruction following the system service request.
2. If the service-specific procedure returned a status other than SS$_WAIT_CALLERS_MODE or SS$_QIO_CROCK, SERVICE_EXIT continues with step 6.
3. If the service-specific procedure returned one of those statuses, SERVICE_EXIT tests the dispatch table entry to see if this service is permitted to return that status. If not, it generates the fatal bugcheck IVSSRVRQST.
4. If the status returned is SS$_WAIT_CALLERS_MODE, SERVICE_EXIT must cause the process to be taken out of execution. (The service-specific procedure has changed the process's state and placed its PCB in the corresponding wait queue, but the process is, in fact, still current and provides the context in which SERVICE_EXIT executes.) Some system services must be reexecuted after the process's wait ends, whereas others must not. SERVICE_EXIT tests the service's dispatch table entry flag SSFLAG_K_WCM_NO_REEXEC to determine which type this service is.

   — If the flag is clear, the service must be reexecuted, and SERVICE_EXIT restores the argument registers from the kernel stack, leaving only the exception frame on the kernel stack. It subtracts 12 from the PC saved in the exception frame, so that it contains the address of the beginning of the system service transfer routine. The process will wait in the access mode from which the service was requested.

     SERVICE_EXIT invokes SCH$WAIT_ANY_MODE, in module SCHEDULER, to take the process out of execution and select a new one to execute.

     Later, when the process becomes current again, it will reexecute the transfer routine and reenter the change mode dispatcher, which will call the service-specific procedure. The Hibernate ($HIBER) system service, for example, uses this mechanism so that a hibernating process can receive ASTs. (Chapter *Scheduling* contains details related to process waits, and Chapter *Process Control and Communication* describes hibernation.)

   — If the flag is set, the service should not reexecute, and SERVICE_EXIT simply invokes SCH$WAIT_ANY_MODE. The process will be placed back into execution at the address following the CALL_PAL CHMK instruction. This mechanism is used only by certain forms of the event flag wait system services.

5. The SS$_QIO_CROCK status is returned only from EXE$QIO, the service-specific procedure for the $QIO system service. It returns that status when a direct I/O buffer page must be faulted into the process's working set. So that outer mode ASTs can be delivered to the process during a potential page fault wait, the page fault must occur in the access mode from which the $QIO service was requested.

   SERVICE_EXIT restores the argument registers from the kernel stack, leaving only the exception frame on the kernel stack. It replaces the PC saved in the exception frame with the address of routine FAULTY$TOWERS, in module SYSTEM_SERVICE DISPATCHER. It executes a CALL_PAL REI instruction, transferring control to FAULTY$TOWERS in the access mode from which the $QIO system service was requested.

   FAULTY$TOWERS accesses the direct I/O buffer page, causing a page fault for that page; any resulting page fault wait occurs in the mode of the $QIO system service requestor. Later, after the page fault completes, the SERVICE_EXIT routine jumps to the $QIO system service transfer routine, which causes reentry to the change mode dispatcher.

6. If the status in R0 is an error or a severe error, SERVICE_EXIT checks whether the process owns any mutexes. In general, a service-specific procedure should release any mutexes that it has acquired before returning to SERVICE_EXIT. To minimize overhead, SERVICE_EXIT only performs the check for mutexes when a service returns an error or a severe error status.

   — If the process owns a mutex, SERVICE_EXIT tests whether the interrupt priority level (IPL) is 2. If so, the assumption is that one system service has acquired a mutex and then requested another system service, which is returning an error status. In this case, SERVICE_EXIT merely executes a CALL_PAL REI instruction to return control to the presumed original service.

   — If the process owns a mutex but is running at IPL 0, SERVICE_EXIT generates a fatal MTXCNTNONZ bugcheck.

   — If the process does not own a mutex, SERVICE_EXIT continues.

   Chapter *Error Handling* describes bugcheck processing and Chapter *Synchronization Techniques* describes mutexes.

7. If system service exceptions are disabled for the access mode in which the system service was requested, SERVICE_EXIT dismisses the CHM*x* exception by executing a CALL_PAL REI instruction, as described in step 1.

8. Otherwise, the process has enabled system service exceptions for the access mode in which the service was requested. (Section 7.8.1 describes the means by which a process requests that errors be signaled as software conditions.) Since the exception code must be entered at IPL 0, SERVICE_EXIT explicitly lowers IPL if the process is running in kernel mode. Executive mode services do not need a similar check because elevated IPL requires kernel mode operation. (Lowering IPL is unnecessary unless the process has enabled system service failure exceptions, because the CALL_PAL REI instruction that dismisses the CHMK exception lowers the IPL.)

9. To signal the system service exception, SERVICE_EXIT transfers control to EXE$SSFAIL, in module EXCEPTION. It signals the condition SS$_SSFAIL to the caller.

   Chapter *Condition Handling* describes condition handling and system service failures.

7.6 Synchronous System and RMS Services

Some system services perform their requested function and always return immediately to their requestor. Others, called asynchronous system services, initiate some system activity on behalf of the requestor and return before the activity is complete. To synchronize with completion of the initiated activity, the requestor can wait for an event flag associated with the system service request. A synchronous service initiates the activity, just as its asynchronous counterpart does, but waits for completion of the activity before returning to its requestor.

A synchronous system service is generally named for the asynchronous system service it requests. A trailing "W" in the name of the synchronous service distinguishes the two: $QIO and $QIOW, for example. Many RMS services also have both synchronous and asynchronous forms. The two forms, however, are not distinct services with different names. Instead, an image requests either asynchronous or synchronous return from a particular RMS service through a bit in the file or record stream data structure associated with the request.

In [*censored*] VMS, a synchronous service is implemented as a mode of caller service that requests the asynchronous service, tests the return status to ensure that the service was initiated, and then waits for the activity initiated by the service to complete. The process waits in the access mode from which it requested the system service. The sections that follow describe both synchronous system services and synchronous returns from RMS services. Although their implementations differ slightly in structure, they are similar in purpose and algorithm.

7.6.1 Synchronous System Services

A VAX/VMS synchronous system service is an inner mode service whose system service vector potentially requests multiple system services: the asynchronous form of the service and, minimally, a wait service. [*Censored*] VMS system service transfer routines are more limited, and the method used to construct a synchronous VAX/VMS system service at service load time is not viable.

Instead, a [*censored*] VMS synchronous system service is implemented as a mode of caller service that performs the same steps as its VAX/VMS counterpart:

1. It requests the asynchronous form of the service.
2. If the asynchronous service returns an error status, the synchronous service returns the error immediately to the service requestor.
3. Otherwise, it branches to a synchronization routine that waits the process in the mode of the system service requestor.

To guarantee completion of a synchronous system service request, the service requestor must specify both an event flag and a status block (I/O status block or lock status block). The asynchronous service-specific procedure clears the event flag and status block associated with the request. The synchronization routine tests the combination of event flag and status block for request completion, placing the process into event flag wait if the request is not complete.

Testing the combination prevents a premature return to the synchronous service requestor as the result of concurrent uses of the same event flag. If the service requestor omits the optional status block, the mechanism reverts to being a simple wait for event flag. The mechanism is requested explicitly as the Synchronize ($SYNCH) system service and implicitly as part of each synchronous system service. Table 7–1 lists the synchronous system services.

The VAX MACRO code for the synchronous service $QIOW is shown in Example 7–1 in a slightly simplified form.

Example 7–1 VAX MACRO Code for the $QIOW System Service

```
EXE$QIOW::
        MOVL    QIO$_EFN(AP),R7   ;Save event flag number for $SYNCH
        MOVL    QIO$_IOSB(AP), R8 ;Save IOSB address, if specified,
                                  ; for $SYNCH
        SUBL    #QIO$_NARGS*4,SP  ;Push arguments on stack for $QIO
        MOVL    R7,QIO$_EFN-4(SP) ; (-4 since no argument count)
        MOVL    QIO$_CHAN(AP),QIO$_CHAN-4(SP)
        .
        .
        .
        MOVL    QIO$_P6(AP),QIO$_P6-4(SP)
        CALLS   #QIO$_NARGS,SYS$QIO
                                  ;Request the $QIO system service
        BLBC    R0,1000$          ;Branch if error queuing request
        BSBW    EXE$SYNCH_LOOP    ;Invoke $SYNCH internal routine
1000$:
        RET                       ;Return to caller
```

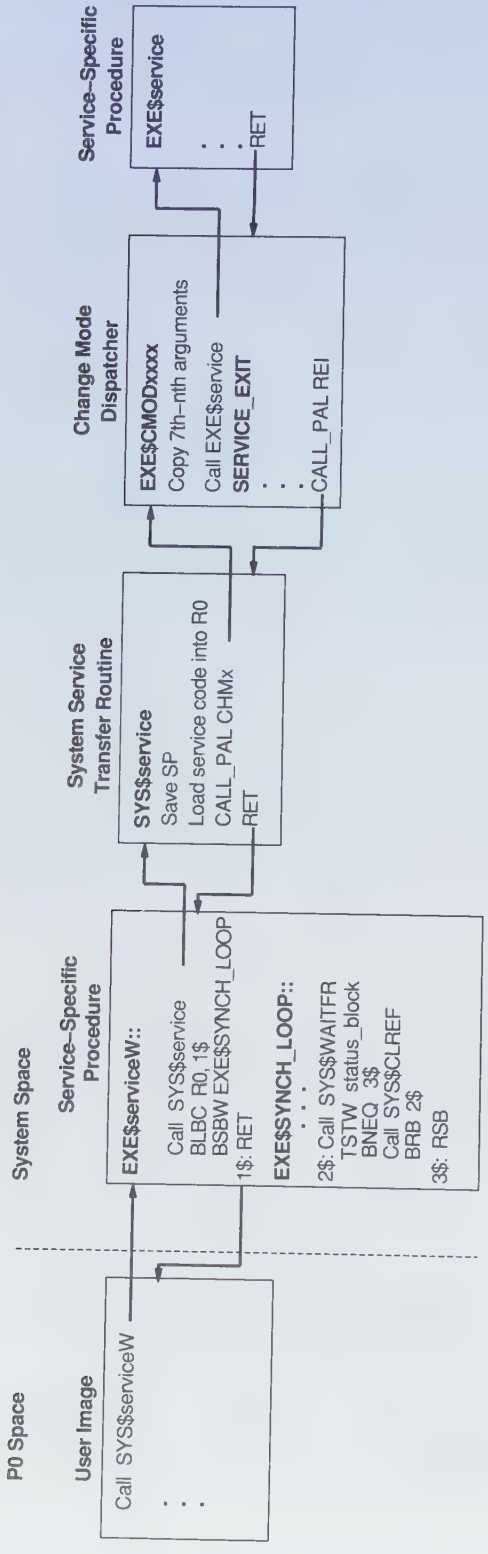
Figure 7–6 illustrates the control flow from a user image, through a service-specific procedure, to the synchronization routine.

The $SYNCH system service procedure, EXE$SYNCH in module SYSSYNCH, executes in the mode of its requestor. It is typically requested with the number of an event flag and the address of a status block. It simply invokes the routine EXE$SYNCH_LOOP, the same routine invoked by synchronous system services such as $QIOW and $ENQW.

EXE$SYNCH_LOOP takes the following steps to determine whether the requested operation is complete:

1. It first tests whether a status block was specified by the requestor. For $GETLKIW and $ENQW, the lock status block serves this purpose; in all other cases, an I/O status block is used.

Figure 7–6 Control Flow of Synchronous Services



If no status block was specified, EXE$SYNCH_LOOP requests the Wait for Single Event Flag ($WAITFR) system service to place the process into an event flag wait state until the specified flag is set. When the flag is set, the process is taken out of its wait state, and EXE$SYNCH_LOOP returns to its invoker, either EXE$SYNCH or a synchronous system service.

2. If a status block was specified, EXE$SYNCH_LOOP tests the status word of the status block. A nonzero status indicates that the asynchronous service has completed, and EXE$SYNCH_LOOP returns to its invoker. Note that if the status block is being used concurrently for multiple requests, one or both of the requested services can complete prematurely; the $SYNCH service assumes that nonzero contents imply asynchronous service completion and does not test whether the event flag is set.

   A zero status indicates that the asynchronous service has not yet completed.

3. EXE$SYNCH_LOOP requests the $WAITFR system service to wait for the specified event flag.

4. When the event flag is set and the process placed into execution, EXE$SYNCH_LOOP tests the low word of the status block. If it is nonzero, EXE$SYNCH_LOOP returns to its invoker.

5. If the low word of the status block is zero, then the flag has been set spuriously, perhaps by another concurrent use. EXE$SYNCH_LOOP requests the Clear Event Flag ($CLREF) system service to clear the flag and retests the low word of the status block in case the request has completed in the meantime. If the word is nonzero, EXE$SYNCH_LOOP requests the Set Event Flag ($SETEF) system service to set the flag that it just cleared and returns to its invoker.

   If the status is still zero, EXE$SYNCH_LOOP proceeds with step 3.

A crucial point in this implementation is that the process waits at the access mode associated with the original synchronous system service request, thus allowing AST delivery to all access modes at least as privileged as that of the synchronous service request. In the usual case where a synchronous system service is requested from user mode, an AST of any access mode can be delivered while the process is waiting for the service to complete.

7.6.2 Synchronous Returns from RMS Services

Often, an RMS service-specific procedure cannot complete an RMS operation until the completion of an I/O or lock request it has made. When an image requests such an RMS service, it can specify that the service procedure not return until the RMS operation is complete. Alternatively, the image can specify that the service return after the I/O or lock request is initiated so that the image can continue execution. Table 7–1 lists the RMS services that can perform synchronous returns.

By setting the asynchronous (ASY) bit in the file access block (FAB) or record access block (RAB), an image indicates that the service should return even though the RMS operation is not complete. When the image can no longer execute without the RMS operation's having been completed, the image requests

the $WAIT RMS service. By default, the bit is clear, and an image requests a synchronous return from an RMS service.

This section describes the mechanism by which an RMS service-specific procedure waits for completion of an I/O or lock request when the ASY bit in the FAB or RAB is clear. It also describes the $WAIT service.

7.6.2.1 RMS Synchronization

When the VAX/VMS change mode dispatcher REIS after dispatching an RMS service that may have to wait, it returns to a stall routine rather than to the RMS service vector. That is, the exception PC has been overwritten with the address of a stall routine.

Because [*censored*] VMS requires nested calls and returns, this method is not viable. Instead, each RMS service that may need to stall the process is implemented as a mode of caller service. The mode of caller service requests an executive mode RMS service. When the mode of caller service returns, it invokes a synchronization routine if a stall is necessary. Although a detailed description of the implementation of RMS services is beyond the scope of this book, the following example illustrates what happens when a process requests an RMS operation and must wait for its completion.

1. RMS$READ in module [RMS]RMS0SERVICES, the $READ service procedure, is a mode of caller service that requests the executive mode service $RMS_READ.

2. RMS$RMS_READ in module [RMS]RMS0BLKIO, the $RMS_READ procedure, requests the $QIO system service on behalf of its caller. It specifies the number of an event flag to be set (one of the flags permanently allocated to RMS) and the address of an executive mode AST procedure to be executed when the I/O operation completes. The AST procedure it specifies is RM$STALLAST, in module [RMS]RM0THDMGR.

   After the $QIO system service returns, the RMS procedure calls RM$STALL, in module [RMS]RM0THDMGR. RM$STALL tests the ASY bit in the FAB or RAB associated with the request. If the bit is clear, RM$STALL saves the state and context of this RMS request in an RMS data structure and returns to the change mode dispatcher with the status RMS$_STALL and the number of the associated event flag.

3. The change mode dispatcher dismisses the CHME exception and returns to the mode of caller RMS service procedure, in this example, RMS$READ.

4. The mode of caller service procedure tests the return status. If the status is anything other than RMS$_STALL, it returns to its caller.

   If the status is RMS$_STALL, it invokes WAIT_IO_DONE, in module [RMS]RMS0SERVICES, to place the process into a wait for the event flag associated with the $QIO request.

5. Executing in the caller's mode, WAIT_IO_DONE sets a "busy" bit in the FAB or RAB associated with the request to ensure that another request cannot lead to the structure's reuse before the wait is over. It then takes steps that parallel those listed for EXE$SYNCH_LOOP in Section 7.6.1. The main difference is that the status tested is in the FAB or RAB associated with the RMS operation.

6. When the I/O request completes, the associated event flag is set and RM$STALLAST is entered. It restores the saved state and context of the RMS request and reenters the original RMS executive mode procedure, in this example, RMS$RMS_READ.

7. RMS$RMS_READ continues execution from the point at which it invoked RM$STALL. If it determines that the RMS operation is not complete, it requests the next service and stalls using the technique just described.

   Otherwise, if the RMS operation is complete, RMS$RMS_READ calls RM$EXRMS, in module [RMS]RM0EXTRMS, to set final status in the associated FAB or RAB and to exit from the executive mode AST procedure.

8. AST exit code returns control to WAIT_IO_DONE, whose event flag wait is now over. Executing in the requestor's access mode, WAIT_IO_DONE checks whether the RAB or FAB status field is zero. If so, it again places the requestor into an event flag wait state. In other words, a nonzero value in the status field of the FAB or RAB is the actual indication that the RMS operation is complete.

   When the status field indicates successful completion or a warning, WAIT_IO_DONE returns to the mode of caller service. Otherwise, when WAIT_IO_DONE discovers an error or the status field indicates an error, it performs the error processing described in Section 7.6.2.3.

A crucial point in this implementation is that the requestor waits at the access mode associated with the original RMS service request and not in executive mode, thus allowing AST delivery to all access modes at least as privileged as that of the service request. In the usual case where an RMS service is requested from user mode, an AST of any access mode can be delivered while the process is waiting for the RMS operation to complete.

7.6.2.2 $WAIT Service

An image requests the $WAIT service to synchronize with completion of an outstanding RMS operation that was requested with an asynchronous return. The service has one argument, the address of the FAB or RAB associated with the outstanding RMS request.

The $WAIT procedure, RMS$WAIT in module [RMS]RMS0SERVICES, executes in the mode of its caller. If, through its actions, the process is placed in a wait state, the process will be able to receive ASTs for its current mode and inner modes. RMS$WAIT requests the executive mode service $RMS_WAIT, passing it the associated structure address.

The $RMS_WAIT service-specific procedure, RMS$RMS_WAIT, in module [RMS]RMS0WAIT, performs several sanity checks on the RMS structure and tests whether the related RMS operation is actually complete. If it is, RMS$RMS_WAIT returns the status of the operation to RMS$WAIT. If not, it clears the event flag associated with the request and returns the status RMS$_STALL to RMS$WAIT along with a flag indicating how the process should be stalled.

If R0 contains the status RMS$_STALL, RMS$WAIT stalls process execution until an asynchronous RMS operation completes. An action flag argument returned by the $RMS_WAIT service determines the method used to decide whether the operation is complete. If the action flag is clear, the completion of the RMS

operation is indicated by the status field in the RAB, so RMS$WAIT invokes WAIT_IO_DONE to stall in the normal manner. Otherwise, RMS$WAIT alone cannot determine completion of the operation. It requests the $WAITFR system service to wait for the event flag specified by the $RMS_WAIT service procedure. When the event flag is set, RMS$WAIT rerequests the $RMS_WAIT service to allow it to decide whether the operation is complete.

7.6.2.3 RMS Error Detection

WAIT_IO_DONE, executing as part of the mode of caller RMS service procedure, reports an error or a severe error by requesting the executive mode RMS service $SSVEXC. The service has one argument, an error status.

Running in executive mode, the $SSVEXC service procedure, RMS$SSVEXC in module [RMS]RMS0SERVICES, simply returns to the change mode dispatcher, passing its one argument as return status.

The change mode dispatcher, whose actions are described in Sections 7.5.2 and 7.5.3, can signal the error as a software condition or simply return it to the image. (Section 7.8.1 describes the means by which a process requests that errors be signaled as software conditions.) The reason for this somewhat roundabout way of returning the error is that it can be signaled as a system service failure software condition only from an inner access mode.

7.7 System Services in Privileged Shareable Images

VMS does not require that all system services be part of an executive image. A user can write system services as part of a privileged shareable image. Moreover, VMS supplies a number of system services in privileged shareable images, including the following:

- $MOUNT, in SYS$SHARE:MOUNTSHR.EXE
- $DISMOU, in SYS$SHARE:DISMNTSHR.EXE
- Services relating to system security, in SYS$SHARE:SECURESHR.EXE and SYS$SHARE:SECURESHRP.EXE
- Services relating to pseudo terminal support, in SYS$SHARE:PTD$SERVICES_SHR.EXE.

Implementing these less frequently used services as privileged shareable images means that they are resident only when explicitly requested and that they are mapped in process space instead of system space. Thus, they page in process working sets rather than in the system working set. This section examines the manner in which control is passed to a system service that is part of a privileged shareable image.

Dispatching to system services in privileged shareable images resembles dispatching to system services in executive images. The basic steps are as follows:

1. An image calls a transfer routine associated with a system service within a privileged shareable image.
2. The transfer routine loads a change mode number into R0 and executes a `CALL_PAL CHMx` instruction.

3. The change mode dispatcher attempts to dispatch to a privileged shareable image whenever R0 contains a value outside the range of those for services in executive images.
4. The change mode dispatcher searches a P1 space data structure to determine which privileged shareable image implements that system service.
5. It then indexes a dispatch table in the privileged shareable image to get the procedure value of the associated service-specific procedure and calls it.

The [*censored*] VMS implementation of privileged shareable images differs from that of VAX/VMS in the following ways:

- The instructions that make up the system service transfer routines associated with a [*censored*] VMS privileged shareable image are written by the image activator rather than by the user.
- A [*censored*] VMS privileged shareable image contains a user-written dispatch table, called a privileged library vector, rather than a dispatching routine.
- [*Censored*] VMS change mode numbers are assigned dynamically by the image activator as a privileged shareable image is activated rather than determined by values specified in the image source.

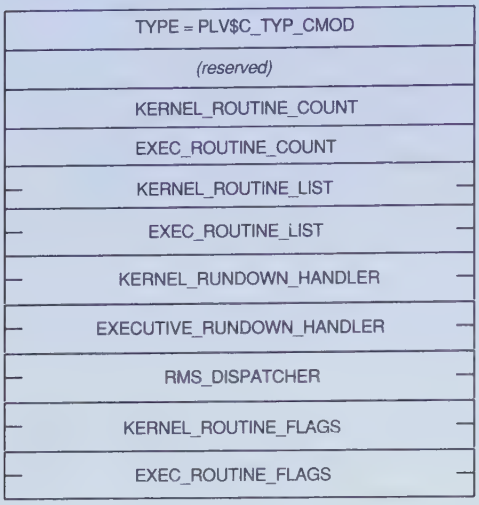
A change mode number for a system service in a privileged shareable image serves only to connect the system service transfer routine's name with a particular dispatch table entry. Change mode numbers in one activated privileged shareable image are independent from those in another and therefore need not be unique within a process. Ambiguity of change mode number values from one privileged shareable image to the next is not a problem, as it would be for VAX/VMS privileged shareable images.

7.7.1 Data Structures Related to Privileged Shareable Images

A privileged shareable image is an image that contains one or more kernel and executive mode system services. Each such privileged shareable image contains a data structure called a privileged library vector (PLV). The STARLET macro $PLVDEF defines symbolic offsets for this structure. The programmer codes the PLV to describe the services within the image and constructs tables containing the addresses of the service-specific procedures. The PLV must be the only contribution to a program section defined with the VEC attribute. Figure 7–7 shows the layout of the PLV.

PLV$L_TYPE identifies the PLV as one that describes inner access mode routines rather than, for example, a message vector. Chapter *Miscellaneous System Services* describes the layout and use of message vector PLVs. This section describes the fields in this type of PLV that are related to system services in a privileged shareable image; chapter *Image Activation and Exit* discusses rundown routines.

Figure 7–7 Layout of a Privileged Library Vector



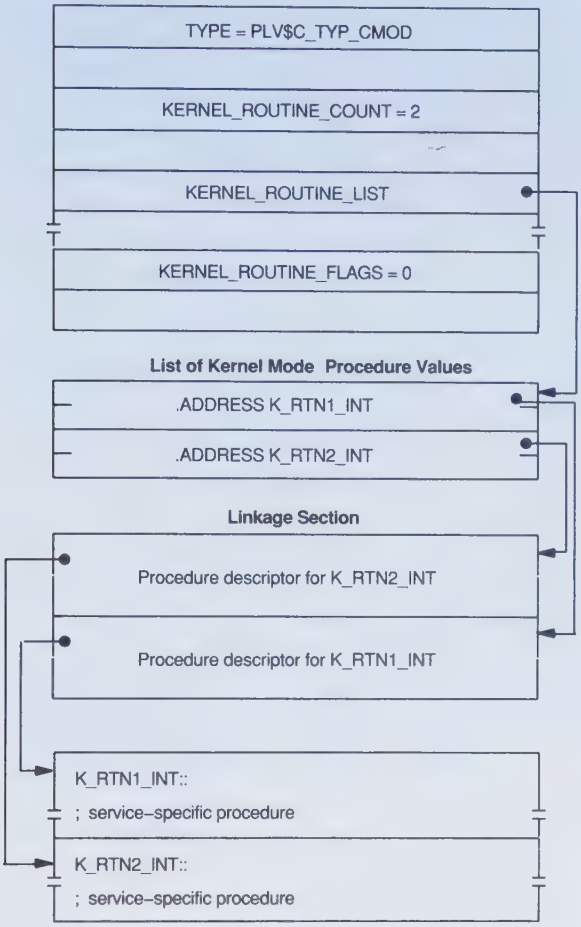
PLV$L_KERNEL_ROUTINE_COUNT contains the number of kernel mode services in the privileged shareable image, and PLV$L_EXEC_ROUTINE_COUNT, the number of executive mode system services. PLV$PS_KERNEL_ROUTINE_LIST and PLV$PS_EXEC_ROUTINE_LIST each point to an array of addresses of service-specific procedure descriptors. Note that a quadword in this data structure is reserved for each pointer, although the pointers themselves each occupy only a longword.

PLV$PS_KERNEL_ROUTINE_FLAGS contains zero or the address of an array of quadwords containing the defined flags associated with each kernel system service. Currently, only two flags are defined, PLV$V_WAIT_CALLERS_MODE and PLV$V_WAIT_CALLERS_NO_REEXEC. When either is set, it means that the kernel mode service may return the status SS$_WAIT_CALLERS_MODE. As described in Section 7.5.1, the two flags differ in specifying whether the system service should be reexecuted after the process's wait ends. Section 7.5.3 describes the change mode dispatcher's reaction to this status. Because there are no defined flags for executive mode, PLV$PS_EXEC_ROUTINE_FLAGS should always contain zero.

Figure 7–8 shows the relations among some of the parts of a privileged shareable image with two kernel mode system services. PLV$PS_KERNEL_ROUTINE_LIST points to the array of kernel service-specific procedure values. Because neither of the services returns SS$_WAIT_CALLERS_MODE, there is no need for a flags table, and PLV$PS_KERNEL_ROUTINE_FLAGS contains zero. Each procedure descriptor contains the address of the code entry point for its service.

After creating one or more modules containing system services and a PLV, the programmer links them to create a privileged shareable image. A privileged shareable image must be linked with the /SHAREABLE qualifier; its image sections that contain inner mode code and data must be protected, through use of either the PROTECT option or the /PROTECT qualifier. In addition, it must

Figure 7–8 Privileged Shareable Image



be linked with the SYMBOL_VECTOR option to force the creation of a symbol vector. The option must specify all global symbols that are to be universal. In particular, it must identify each system service. Each system service must be identified in that option as a procedure.

The format of the SYMBOL_VECTOR option, for example, to describe the procedures in the example of Figure 7–8 is as follows:

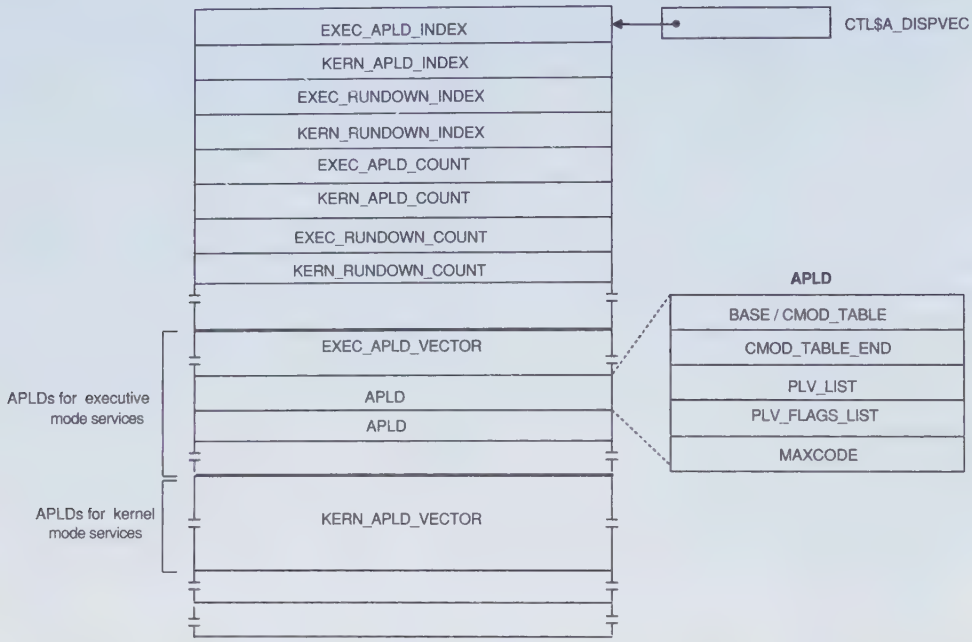
```
SYMBOL_VECTOR = (-
                 K_RTN1_EXT/K_RTN1_INT = PROCEDURE,
                 K_RTN2_EXT/K_RTN2_INT = PROCEDURE)
```

In this example, each procedure is identified by both an externally visible name, which is the universal alias name, and the internal name of the service-specific procedure. Code within the privileged shareable image can call the procedure through its internal name or its external name. It must use the external name if the transfer routine must be executed to change access mode. Otherwise, if no access mode change is required, the internal name may be used for a more direct transfer of control.

The shareable image must be installed with the /PROTECTED and /SHARED qualifiers.

An executable image can be linked with multiple privileged shareable images. A compound data structure called the activated privileged library dispatch vector contains information about the privileged shareable images that are active in the process. Figure 7–9 shows some of the fields in this data structure, whose starting address is contained in P1 space location CTL$A_DISPVEC. The macro $APLDDEF defines symbolic offsets for fields in this structure.

Figure 7–9 Layout of an Activated Privileged Library Dispatch Vector



The structure includes two tables, one for executive mode and one for kernel mode. The executive mode table begins at offset APLD$R_EXEC_APLD_VECTOR, and the kernel mode, at offset APLD$R_KERN_APLD_VECTOR. When a privileged shareable image is activated that contains inner mode system services, a substructure called an activated privileged library dispatch vector entry (APLD) is initialized in the executive or kernel mode table. If the image contains both executive and kernel mode services, an APLD is initialized in both tables. Each table can accommodate up to 42 APLDs. The image activator initializes an APLD from information in the PLV of the privileged shareable image being activated.

An individual APLD describes the set of executive mode or kernel mode system services within one activated privileged shareable image. APLD$PS_CMOD_TABLE contains the address of the beginning of the system service transfer routines associated with the image, and APLD$PS_CMOD_TABLE_END contains the address of the end of the transfer routines. If this APLD describes executive mode services, APLD$PS_PLV_LIST contains the address of the list of executive mode service-specific procedures; otherwise, it contains the address of the list of kernel mode service-specific procedures. APLD$PS_PLV_FLAGS_LIST contains the address of the flags quadwords for the services. APLD$L_MAXCODE contains the value of the highest change mode number assigned to a service of that particular mode.

APLD$L_EXEC_APLD_INDEX contains the number of executive mode APLDs that have been initialized, and APLD$L_KERN_APLD_INDEX, the number of kernel mode APLDs. As the image activator performs fixup and relocation on a privileged shareable image, it increments APLD$L_EXEC_APLD_COUNT, APLD$L_KERN_APLD_COUNT, or both.

The structure also describes rundown routines and message vectors associated with activated privileged shareable images. Chapter *Image Activation and Exit* describes the image activator and the use of rundown routines, and Chapter *Miscellaneous System Services,* the use of message vectors.

When a user runs an executable image that has been linked with a privileged shareable image like the one shown in Figure 7–8, the image activator activates both images. The presence of the VEC image section in the privileged shareable image causes the image activator to perform additional processing for the privileged shareable image.

The image activator takes the following additional steps:

1. It initializes an APLD in the kernel mode table in the activated privileged library dispatch vector, copying information from the PLV to the APLD.
2. It allocates additional P0 space to build a system service transfer routine and procedure descriptor for each system service in the privileged shareable image and stores the low and high addresses of the transfer routines in APLD$PS_CMOD_TABLE and APLD$PS_CMOD_TABLE_END.

   The transfer routine for each system service resembles those discussed in Section 7.2.3. One key difference is that the image activator rather than EXE$CONNECT_SERVICES assigns the change mode number. The image activator assigns the number 65,536 * 1 (equal to 1 in the high-order word of a longword) to the first kernel mode service and the first executive mode service in each privileged shareable image it activates. The next number in the sequence is 65,536 * 2. The sequence continues with increasing multiples of 65,536.

   The other key difference between this transfer routine and those previously discussed is the instruction that loads the change mode number into R0. A multiple *n* of 65,536 is the integer *n* shifted into the high-order word of the longword; such a number can be loaded into R0 with the instruction `LDAH R0,n(R31)`. Thus, system service transfer routines are the same length whether the service is part of an executive image or a privileged shareable image.

3. The image activator searches the privileged shareable image's symbol vector to locate each linkage pair that contains one of the procedure values in the APLD$PS_PLV_LIST. It modifies the linkage pair so that it describes the procedure descriptor and code entry point addresses of the corresponding system service transfer routine.
4. It establishes a more restrictive protection, user-read and executive-write (UREW), on protected image section pages, namely, the pages that contain the privileged shareable image, its system service transfer routines, and their procedure descriptors.

Figure 7–10 shows the activated form of the privileged shareable image of Figure 7–8. Chapter *Image Activation and Exit* contains details on image activation.

7.7.2 Dispatching to System Services in a Privileged Shareable Image

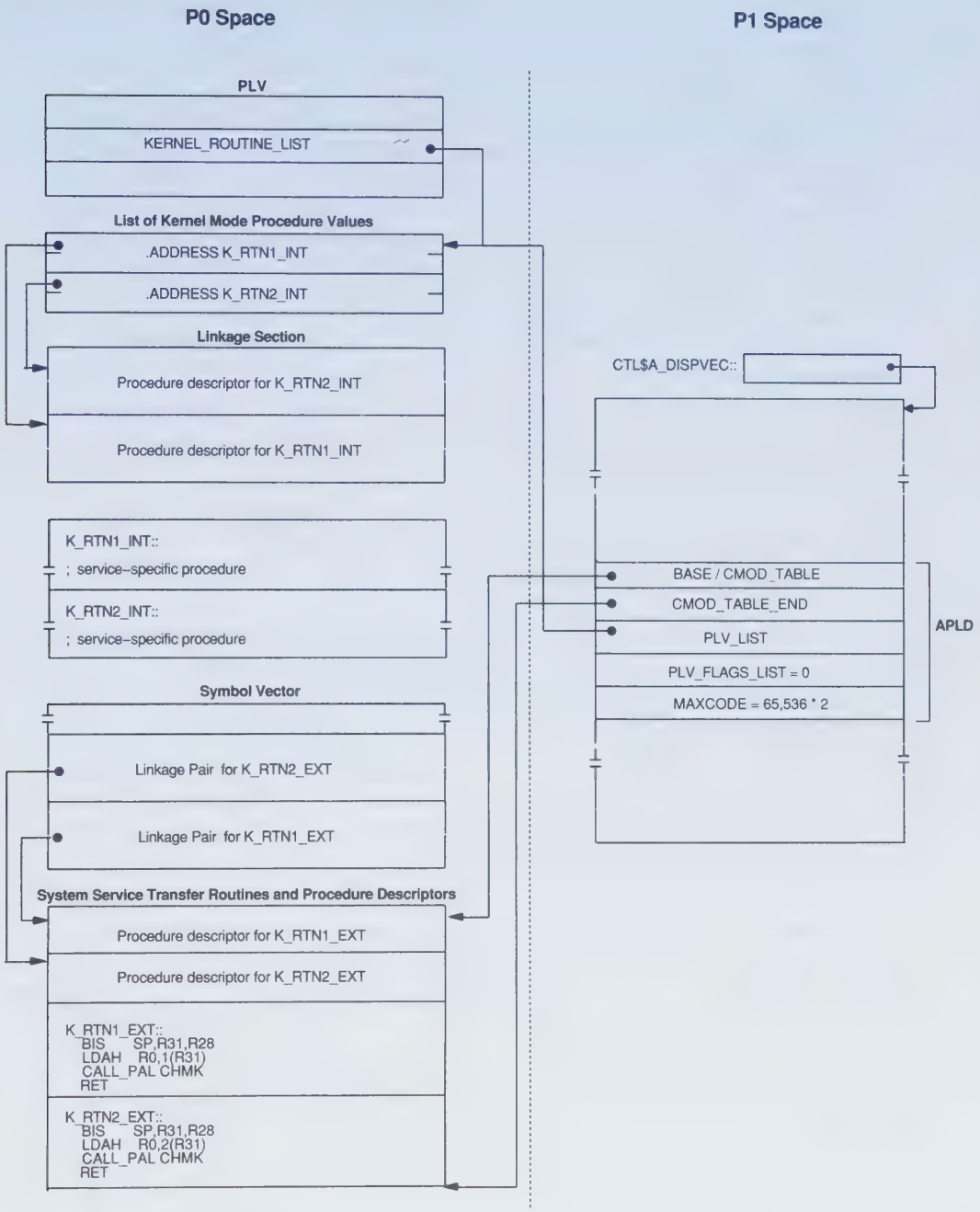
When an image requesting a system service in a privileged shareable image is activated, the image activator updates its linkage pair that represents the service procedure with information from the shareable image's symbol vector. The linkage pair is updated to contain the addresses of the transfer routine code entry and procedure descriptor.

When the image calls the procedure, it transfers control to the corresponding system service transfer routine, which loads the corresponding change mode number into R0 and executes a `CALL_PAL CHMx` instruction. Executing that instruction causes an exception, which is dispatched to the appropriate change mode dispatcher. The PC saved in the exception frame is an address within the system service transfer routine.

For any CHMK or CHME exception, each change mode dispatcher performs some initial operations, such as checking the accessibility of arguments in memory, and, for kernel mode system services, storing the PCB address in R4 (see Section 7.5.2). When a change mode dispatcher detects that the change mode number is equal to or greater than the value in CMOD$GL_CHM*x*_LIMIT, it tries to dispatch to a system service in a privileged shareable image. The description that follows applies to both the executive mode and kernel mode dispatchers unless otherwise noted.

1. The change mode dispatcher checks that the change mode number is 65,536 or larger, returning the error status SS$_ILLSER if not.
2. From the exception frame on the stack, it gets the PC, the address of the instruction following the `CALL_PAL CHMx` instruction in the system service transfer routine.
3. It examines the initialized APLDs in either the executive or kernel mode part of the process's activated privileged library dispatch vector. It scans until it reaches the end or finds one whose set of system service transfer routines contains the address of the change mode instruction that triggered this exception. APLD$PS_CMOD_TABLE and APLD$PS_CMOD_TABLE_END point to the low- and high-address ends of these transfer routines.

Figure 7–10 Activated Privileged Shareable Image



If it reaches the end without finding such an APLD, the change mode dispatcher returns the error status SS$_ILLSER.

4. If the change mode dispatcher finds an APLD whose transfer routines contain the exception PC, it compares the change mode number to the contents of APLD$L_MAXCODE. If the change mode number is larger, the dispatcher returns the error status SS$_ILLSER.

5. It transforms the change mode number into a number suitable for use as a quadword index into the list of procedure descriptors in the PLV. This requires the combination of shifting it 13 bits to the right (to divide by 65,536 and then to multiply by 8) and then subtracting 8, since the list of procedure descriptors is zero-based.
6. The kernel mode dispatcher indexes into the PLV flags table, pointed to by APLD$PS_PLV_FLAGS_LIST, to test whether this service can return SS$_WAIT_CALLERS_MODE. If so, it copies the AI and argument registers (up through the first six arguments) to the kernel stack. Each dispatcher performs the same argument list processing that it does for a system service in an executive image. Section 7.5.2 describes processing of the argument list.
7. The change mode dispatcher indexes into the PLV list of procedure values, pointed to by APLD$PS_PLV_LIST, to get the address of the procedure descriptor for this service.
8. It calls the service-specific procedure.
9. When the service-specific procedure returns, the change mode dispatcher jumps to SERVICE_EXIT.

7.8 Related System Services

VMS provides several system services that are closely related to system service dispatching and the change mode instructions. Chapter *Condition Handling* describes the Declare Change Mode Handler ($DCLCMH) system service. This section describes the Set System Service Failure Exception Mode ($SETSFM) system service, the change mode system services, and the Set System Service Filter ($SETSSF) system service.

7.8.1 $SETSFM System Service

The $SETSFM system service enables or disables the signaling of errors and severe errors returned by an inner mode service-specific procedure. (A mode of caller service always simply returns to its requestor.) By default, an error or severe status is merely returned to the system service requestor in R0. If system service failure exception mode is enabled, SERVICE_EXIT instead signals an error or severe error as a software condition. The condition is signaled in the mode from which the system service was requested. Chapter *Condition Handling* contains more details on condition signaling and handling.

System service failure exception mode is controlled by four bits, one for each access mode, in PCB$L_STS. The bits begin at bit position PCB$V_SSFEXC. A bit, when set, enables signaling of an error or severe error. By default, all four bits are clear.

The $SETSFM procedure, EXE$SETSFM in module SYSSETMOD, simply sets or clears the bit in PCB$L_STS corresponding to the access mode from which the system service was requested and returns the status SS$_WASSET or SS$_WASCLR to describe the previous state of the bit.

7.8.2 Change Mode System Services

The Change to Kernel Mode ($CMKRNL) and Change to Executive Mode ($CMEXEC) system services provide a simple path for privileged processes to execute code in kernel or executive mode. Their service-specific procedures are EXE$CMKRNL, which runs in kernel mode, and EXE$CMEXEC, which runs in executive mode, both in module SYSCHGMOD. Each procedure checks that the process has the necessary privilege (CMKRNL or CMEXEC) and, if not, returns the status SS$_NOPRIV. Note that if $CMKRNL is requested from executive mode, no privilege check is made.

Each of these system services has one required argument, the address of a user-specified procedure, and one optional argument, the address of a list of arguments for that procedure. If the second argument is nonzero, it must be the address of an argument list that begins with an argument count. EXE$CMKRNL and EXE$CMEXEC call the user-specified procedure, which must load a status into R0 before returning. EXE$CMKRNL and EXE$CMEXEC return R0 unchanged to the change mode dispatcher, which uses its contents to determine whether an error occurred.

When the user-specified procedure returns, the system service procedure returns to SERVICE_EXIT. Section 7.5.3 describes the actions taken by the change mode dispatcher when control returns to it.

7.8.3 System Service Filtering

Some applications (especially user-written command language interpreters) require that images running in user mode have no direct access to system and RMS services. VMS provides a mechanism whereby code running in an inner mode can prevent user mode code in that process from directly requesting certain services. The mechanism is known as system service filtering. It includes alternative system service dispatchers, a SYSGEN parameter to enable system service filtering systemwide, and the system service $SETSSF to enable filtering of a particular process's user mode system service requests.

Each VMS system service in an executive image specifies an inhibit mask at assembly time as a parameter to the SYSTEM_SERVICE macro. The inhibit mask indicates whether execution of the system service can be blocked by the $SETSSF system service. If the service can be blocked, the mask also indicates the system service filter groups for which the service is blocked. Group 0 specifies all services except $EXIT; group 1 specifies most services, with the exception of $EXIT and those services required for condition handling or image rundown. The *VMS System Services Reference Manual* lists the services that are not blocked by $SETSSF.

The mask is stored in the system service descriptor block for the service. As a service is loaded, its inhibit mask is copied from its descriptor block into one of two tables, depending on the mode of the service. CMOD$AB_KERNEL_INHIBIT_MASK is the table for kernel mode services, and CMOD$AB_EXEC_INHIBIT_MASK for executive mode. Each table is indexed by change mode number; for example, the kernel mode system service assigned change mode number *x* stores its inhibit mask at offset *x* from the address in CMOD$AB_KERNEL_INHIBIT_MASK.

The byte at offset CTL$GB_SSFILTER in the per-process control region contains the system service filter mask for a particular process. Usually this mask contains the value zero. The $SETSSF system service procedure, EXE$SETSSF in module SYSSETSSF, confirms that it was requested from a mode other than user and, if not, returns the error status SS$_NOPRIV. Otherwise, it writes the mask value specified as its argument into CTL$GB_SSFILTER.

The bit EXE$V_SSINHIBIT at global location EXE$GL_DEFFLAGS corresponds to the SYSGEN parameter SSINHIBIT, which, when set, enables system service filtering. If system initialization code discovers that the inhibit bit is set, it loads the SCB entries for CHME and CHMK exceptions with the addresses of the alternative dispatchers EXE$CMODEXECX and EXE$CMODKRNLX, in module SYSTEM_SERVICE_DISPATCHER.

The processor dispatches to these alternative change mode dispatchers when CHME and CHMK exceptions occur. They branch to the standard change mode dispatchers for CALL_PAL CHM*x* instructions executed in inner modes. However, for a CALL_PAL CHM*x* instruction executed in user mode, the alternative dispatcher ANDs the value in CTL$GB_SSFILTER with the value in the appropriate system service filter table (CMOD$AB_EXEC_INHIBIT_MASK or CMOD$AB_KERNEL_INHIBIT_MASK) entry indexed by the CHM*x* number. If the result of the AND is zero, the dispatcher branches to the standard change mode dispatcher. If the result of the AND is nonzero, the dispatcher returns the error status SS$_INHCHME or SS$_INHCHMK, depending on the mode of the system service.

If CTL$GB_SSFILTER is nonzero, the dispatcher also denies access to services in privileged shareable images. An attempt to request those services results in the error SS$_INHCHME or SS$_INHCHMK, depending on the mode of the service.

7.9 Revision Information

Source Module	Build Date
[SYS]SYSTEM_SERVICE_DISPATCHER.M64	4-Nov-1991
[LIB]APLDDEF.SDL	16-Sep-1991
[STARLET]PLVDEF.SDL	15-Nov-1991
[SYS]SYSCHGMOD.MAR	16-Sep-1991
[SYS]CHANGE_MODE_DATA.MAR	16-Sep-1991
[SYS]SYSSETMOD.MAR	16-Sep-1991
[SYS]SYSSETSSF.MAR	16-Sep-1991
[SYS]SYSSYNCH.MAR	16-Sep-1991
[RMS]RMS0SERVICES.B32	16-Sep-1991

Chapter 8

ASTs

What you want, what you're hanging around in the world
waiting for, is for something to occur to you.

Robert Frost

An asynchronous system trap (AST) is a mechanism that enables an asynchronous event to trigger a change in the control flow within a process. Specifically, as soon as possible after the asynchronous event occurs, a procedure or routine designated by either the process or the system executes in the context of the process.

A process can request an AST as notification that an asynchronous system service has completed. ASTs requested by the system result from operations such as I/O postprocessing, process suspension, and process deletion. These operations require that VMS executive code execute in the context of a specific process. ASTs fulfill this need.

When the asynchronous event occurs, the executive inserts an AST control block (ACB) that describes the AST on a process-specific queue. The process eventually becomes current, and an AST delivery interrupt is triggered. Executing in the context of that process, an AST delivery interrupt service routine removes the ACB from the queue and dispatches into the AST procedure it describes.

This chapter discusses the [*censored*] architectural support for ASTs and the [*censored*] VMS implementation of ASTs, in particular the queuing of ACBs and delivery of ASTs. Based on the VAX/VMS implementation of ASTs, the [*censored*] VMS implementation preserves documented external VMS interfaces. The chapter also describes AST-related system services and some examples of VMS executive use of ASTs.

8.1 Characteristics of ASTs

An AST is ranked by its access mode, the mode in which the AST procedure executes. In general, within a process, inner access mode activity is higher priority than outer access mode activity. For ASTs in particular, delivery of an inner mode AST is more important than delivery of an outer mode AST. Moreover, an inner access mode thread of execution is never interrupted for delivery of an AST to a less privileged mode.

Within an access mode, ASTs are serialized. The first to be queued is the first to be delivered. Only one AST of a particular access mode is active at one time; the executive does not interrupt an AST thread to deliver another AST to the same access mode. This serialization limits the number of concurrent threads of execution within a process and helps ensure that AST procedures are not entered recursively, thus simplifying synchronization among the different threads in an access mode.

Running in a particular access mode, a process can enable or disable AST delivery to that mode. Temporarily disabling AST delivery enables a non-AST thread of execution to synchronize access to a data structure shared with an AST thread by blocking execution of the AST. The concept of AST reentrancy and ways of achieving it are described in the *Guide to Creating VMS Modular Procedures*.

Kernel mode ASTs are further divided into normal and special. A special kernel mode AST is one requested by the system, and its execution is considered more critical than the execution of a normal kernel mode AST. Thus, its delivery cannot be disabled, although its delivery can be blocked temporarily while the process executes at interrupt priority level (IPL) 2 or above.

8.2 Architectural Support for ASTs

[*Censored*] hardware and privileged architecture library (PAL) instructions and routines assist VMS in the queuing and delivery of ASTs by providing the following:

- The AST summary (ASTSR) and AST enable (ASTEN) processor registers, and their save areas in the hardware privileged context block (HWPCB)
- The AST enable (`CALL_PAL SWASTEN`) instruction
- Support in several PALcode routines to detect that an AST interrupt should be requested
- The four IPL 2 AST interrupts, whose service routines are described in Section 8.7.1

The hardware context of a [*censored*] process includes four AST summary bits in the ASTSR processor register and four AST enable bits in the ASTEN processor register. In each register there is a bit for each access mode; the lowest order bit corresponds to kernel mode. The registers have a combined save area in the HWPCB, at field PHD$Q_ASTSR_ASTEN. The SWPCTX PALcode routine stores these registers in the HWPCB of the process being taken out of execution and loads them from the HWPCB of the process being placed into execution. Chapter *Scheduling* contains more information on the HWPCB and swapping of process context.

The ASTSR register summarizes the state of the process's AST queues. An AST summary bit, when set, means that at least one AST at that mode is pending; a pending AST is an ACB representing a particular AST queued to the process. Running in kernel mode, the executive can read and change the state of the process's summary bits through the `CALL_PAL MFPR_ASTSR` and `CALL_PAL MTPR_ASTSR` instructions.

An AST enable bit, when set, means that AST delivery to that access mode is enabled. When a process is created, all four bits are set to enable delivery to the respective modes. A process can set or clear the enable bit for the current mode by executing the CALL_PAL SWASTEN instruction. Section 8.4 provides more details on enabling and disabling AST delivery. Running in kernel mode, the executive can read and modify the state of any of the process's enable bits through the CALL_PAL MFPR_ASTEN and CALL_PAL MTPR_ASTEN instructions. The executive executes the CALL_PAL MTPR_ASTEN instruction when an image is run down, to enable AST delivery to the mode of the rundown and less privileged modes. Chapter *Image Activation and Exit* describes image rundown.

An AST interrupt is requested for a particular access mode when all the following conditions are true:

- The bit corresponding to the mode is set in the ASTSR register, indicating that an AST of that mode is pending.
- The bit corresponding to the mode is set in the ASTEN register, indicating that AST delivery to that mode is enabled.
- The process is executing in a mode equal to or less privileged than that of the pending AST.
- IPL is less than 2.

AST interrupt requests are triggered by the PALcode routines that implement instructions that alter one or more of these conditions. These instructions are REI (decreasing access mode and IPL), MTPR_IPL (decreasing IPL), SWASTEN (enabling AST delivery), MTPR_ASTEN (enabling AST delivery), and MTPR_ASTSR (declaring a pending AST). Each of these routines, when otherwise complete, checks the previously listed conditions to determine whether there is a deliverable AST. If so, the routine triggers an AST interrupt request. [*Censored*] hardware or PALcode, depending on the implementation, clears the ASTSR bit and requests the AST interrupt.

Typically, the ASTSR register is modified several times during the queuing and delivery of an AST:

- When an AST is queued to a current process, the bit corresponding to its mode is set.
- When an AST interrupt is initiated for a particular access mode, the bit corresponding to the mode has been cleared by the hardware or PALcode that requested the interrupt. Otherwise, executing any of the previously mentioned PALcode instructions while the AST is active would result in a redundant AST interrupt to the same access mode.
- When an AST procedure is done and exits, the register is updated to reflect other potential pending ASTs for the same access mode. Section 8.3.2 describes the algorithm for updating the register.

The PALcode routines that modify the ASTSR and ASTEN registers accept an argument composed of two masks: one containing the bits to be left unchanged and the other the bits to be explicitly set or cleared. This enables kernel mode code to specify that certain bits be changed and others simply preserved without having to read the state of the register first.

The STARLET macro $PRDEF defines symbolic values for the bit positions within each register and symbols for values to load into them. For example, the symbol PR$M_ASTSR_KPD can be ORed with PR$M_ASTSR_PRSRV_ALL to construct an eight-bit mask to set kernel mode pending and preserve any other set pending bits.

8.3 AST Data Structures

This section describes the data structures related to ASTs. The principal structure is the ACB, described in Section 8.3.1. The executive queues ACBs to a process as the corresponding asynchronous events (for example, I/O completion and timer expiration) occur.

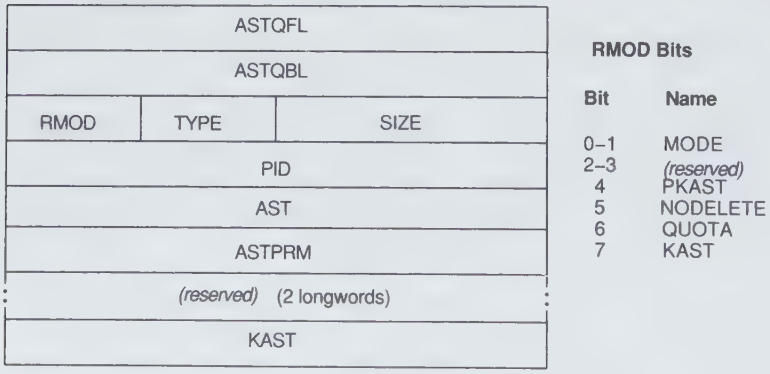
The ACB listheads for a process are in its software process control block (PCB), which also contains several other fields related to ASTs. Section 8.3.2 contains further information.

In addition to the PHD$Q_ASTSR_ASTEN field already described, the process header (PHD) has other AST-related information. The field PHD$W_ASTLM contains the process's authorized AST quota, typically taken from the user authorization file. The PHD$L_FLAGS2 bit PHD$V_AST_PENDING, when set, means that a deliverable AST has been queued to the process. The scheduler tests the flag when it places a process into execution to optimize delivery of the AST (see Section 8.7).

8.3.1 AST Control Block

An ACB describes a pending AST request and identifies the target process, the procedure to be executed, and the access mode of the procedure. Figure 8–1 shows the layout of an ACB. The macro $ACBDEF defines symbolic names for the fields in an ACB.

Figure 8–1 Layout of an AST Control Block



An ACB is allocated from nonpaged pool, often as part of a larger structure associated with the requested asynchronous event. An ACB is included as the first section of an I/O request packet (IRP), lock block (LKB), and timer queue entry (TQE). (In fact, the two reserved longwords in an ACB provide space necessary to accommodate IRP fields.)

ACB$L_ASTQFL and ACB$L_ASTQBL link the ACB into one of the AST queues in the PCB.

The field ACB$B_RMOD contains five bit fields:

- Bits <0:1> (ACB$V_MODE) specify the access mode at which the AST procedure is to execute.
- Bit <4> (ACB$V_PKAST), when set, indicates the presence of a piggyback special kernel mode AST (see Section 8.8.4).
- Bit <5> (ACB$V_NODELETE), when set, indicates that the ACB should not be deallocated after the AST is delivered.
- Bit <6> (ACB$V_QUOTA), when set, indicates that the process AST quota has been charged for this ACB.
- Bit <7> (ACB$V_KAST), when set, indicates the presence of a system-requested special kernel mode AST (see Section 8.8). If ACB$V_KAST is clear, the ACB describes a normal AST.

ACB$L_PID contains the internal process ID of the process that is to receive the AST.

ACB$L_AST and ACB$L_ASTPRM contain the procedure value of the designated AST procedure and its optional argument.

ACB$L_KAST contains the procedure value of a system-requested special kernel mode AST routine if the ACB$V_PKAST or ACB$V_KAST bit of ACB$B_RMOD is set.

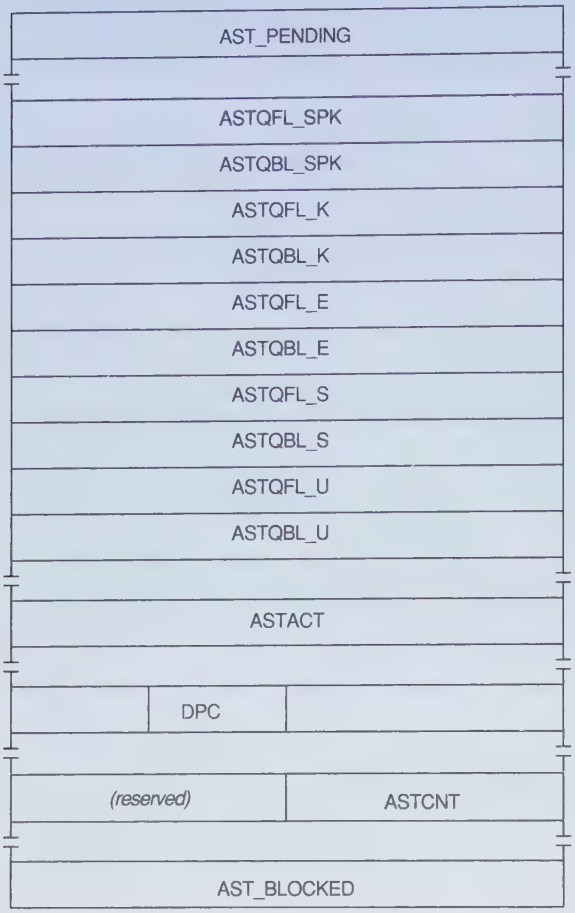
8.3.2 PCB Fields Related to ASTs

Figure 8–2 shows the PCB fields related to AST queuing and delivery.

Each process has five absolute queues of pending ASTs: one for each access mode and one for special kernel mode ASTs. The listheads of these queues are in the PCB. The ACBs in each queue are in the chronological order in which the ASTs were queued to the process. The fields PCB$L_ASTQFL_SPK and PCB$L_ASTQBL_SPK are the listhead for special kernel mode ACBs queued to the process. The other listheads' field names are PCB$L_ASTQFL_*x* and PCB$L_ASTQBL_*x*, where *x* is a letter indicating the access mode.

Because executive routines access these queues holding the SCHED spinlock, they have no need to perform atomic INSQUES and REMQUES. Instead, some of them insert and remove ACBs through sequences of instructions that outperform the equivalent PALcode routines. Entering PALcode would flush the instruction pipeline. Moreover, the PALcode routines take extra steps to provide atomicity guarantees that are unnecessary under these circumstances.

Figure 8–2 AST-Related Fields in the Software PCB



The condition (empty or not) of each of these queues is summarized by one of the low-order four bits in PCB$L_AST_PENDING. If the queue contains one or more ACBs, the bit is set; if the queue is empty, the bit is clear. Bit 0 represents both the special kernel and kernel queues. The appropriate bit in PCB$L_AST_PENDING is set by SCH$QAST, in module ASTDEL, when it queues an AST to a process. When the AST delivery interrupt service routine removes the last AST from a PCB queue during AST delivery, it clears the corresponding bit in PCB$L_AST_PENDING. When the AST procedure is complete, AST exit code copies this longword to update the ASTSR register rather than acquiring the SCHED spinlock and examining each ACB queue to determine which bits to set. Section 8.2 lists circumstances under which the ASTSR register is updated.

The field PCB$W_ASTCNT specifies how many concurrent ASTs the process can request at the moment. It is initialized to the process's AST quota from PHD$W_ASTLM. When a process requests an asynchronous system service with AST notification or when a process declares an AST by requesting the Declare AST ($DCLAST) system service, the system service confirms that PCB$W_ASTCNT is greater than zero and then decrements it, to charge the process AST quota. If the count is not greater than zero, the service returns the error

SS$_EXQUOTA. PCB$W_ASTCNT is incremented and decremented atomically with the load-locked/store-conditional mechanism described in Chapter *Synchronization Techniques*.

It is the responsibility of the system service and of any other code charging AST quota to set the ACB$V_QUOTA bit in the ACB (see Section 8.3.1) as a flag that quota must be restored when this AST is delivered. When such an AST is delivered, the AST delivery routine increments PCB$W_ASTCNT. The difference between the contents of PHD$W_ASTLM and PCB$W_ASTCNT is the number of outstanding ASTs the process has requested.

The process delete pending count, PCB$B_DPC, is incremented for every reason the process should not be deleted or suspended. It is incremented by the Files-11 Extended QIO Processor (XQP) to indicate that an XQP operation is in progress and that the process should not be deleted or suspended until the operation completes. This field is modified and tested only within process context. Section 8.9 discusses the use of this field and its significance to ASTs in more detail.

The low-order four bits of PCB$B_ASTACT specify in which modes ASTs are active. One bit corresponds to each access mode, with bit 0 for kernel mode. Each PCB$B_ASTACT bit, when set, indicates that an AST is active at that access mode in the process. Note that these bits describe only normal ASTs; that is, the low bit of PCB$B_ASTACT does not describe special kernel mode AST activity.

The executive uses these bits to serialize ASTs for each access mode. The AST delivery routine tests and sets the bit corresponding to the mode of the interrupt to guarantee that only one AST for a given access mode executes at one time. When the AST is complete, AST exit code clears the bit. It is possible, though not usual, for an AST procedure itself to clear a PCB$B_ASTACT bit using the Clear AST ($CLRAST) system service (see Section 8.7.3). (To generate simpler code, the field has been extended to a longword and is also called PCB$L_ASTACT.)

The low-order bit in PCB$L_AST_BLOCKED temporarily records the state of the kernel mode AST enable bit. The *censored* architecture does not distinguish between normal kernel mode ASTs and special kernel mode ASTs; in particular, there is only one kernel mode enable bit. Nonetheless, when a process disables delivery of normal kernel mode ASTs, special kernel mode ASTs must still be deliverable. Therefore, when a special kernel mode AST is queued to a process that has disabled kernel mode delivery, the executive reenables kernel mode delivery and sets the low-order bit of PCB$L_AST_BLOCKED. After the last special kernel mode AST is delivered, the disabled state of the kernel mode enable bit will be restored.

Accesses to the ACB queues and to the fields PCB$L_AST_PENDING and PCB$L_AST_BLOCKED are synchronized with the SCHED spinlock. PCB$B_ASTACT is accessed only from within process context: bits in it are tested and set at IPL 2 in the AST delivery interrupt service routine and cleared at IPL 0 by AST exit code.

8.4 Enabling and Disabling AST Delivery

VAX/VMS support for enabling and disabling ASTs is implemented entirely in software. A process sets or clears AST enable bits in the PCB by requesting the Set AST Enable ($SETAST) system service. Before dispatching to an AST procedure, the AST delivery interrupt service routine must check that delivery to that mode is currently enabled and, if it is not, dismiss the interrupt.

In contrast, AST enable bits are part of [*censored*] hardware process context. A process running in any access mode can simply execute the CALL_PAL SWASTEN instruction; this instruction provides a fast way to enable or disable delivery to the current mode. Code running in kernel mode can execute the CALL_PAL MTPR_ASTEN instruction to affect any modes. A PALcode routine that detects the need to request an AST interrupt itself tests the enable bit for that mode; AST delivery interrupts are not requested for disabled modes.

For VAX/VMS compatibility, [*censored*] VMS supports the $SETAST system service, implementing it as a mode of caller service.

EXE$SETAST, the $SETAST system service procedure, simply executes a CALL_PAL SWASTEN instruction with the contents of the ENBFLG argument. EXE$SETAST then returns either the status SS$_WASCLR or SS$_WASSET to reflect the original state of the AST enable bit. Enabling delivery of ASTs to a mode for which there is one or more pending ASTs results in an immediate IPL 2 AST delivery interrupt request. That is, the AST is delivered before control returns to the instruction following the CALL_PAL SWASTEN instruction in the system service.

Temporarily disabling AST delivery enables a non-AST thread of execution to synchronize access to a data structure shared with an AST thread at the same access mode by blocking execution of the AST.

8.5 Creating an AST

ASTs can be created in three ways. The first is a process request for AST notification of the completion of an asynchronous system service, such as Queue I/O Request ($QIO) or Enqueue Lock Request ($ENQ). The arguments for these system services include an AST procedure value and an argument to be passed to the AST procedure. The system service charges the process AST quota and allocates an ACB. When the asynchronous part of the service is complete, the service queues the ACB to the process.

The second is the system's queuing an AST to execute code in the context of the selected process. An ACB used in this situation is not deducted from the AST quota of the target process because of its involuntary nature; the ACB$V_QUOTA bit is clear to indicate this.

The system's ability to initiate the execution of code in a particular process context is crucial to VMS operations. Only the AST mechanism provides this capability. The executive employs this mechanism primarily to access the process's virtual address space.

In a virtual memory operating system such as VMS, resolving a per-process address outside of its process context is difficult at best. The process's pages, as well as page table pages, might not be resident; they can be in a page file, swap file, or in transition. Rather than attempt to locate the relevant page table page(s) and process page(s), VMS resolves the address in process context through the AST mechanism so that standard memory management mechanisms can be used.

Examples of the system's queuing an AST include the following:

- I/O postprocessing
- The Force Exit ($FORCEX) system service
- Expiration of CPU time quota
- Working set limit adjustment performed as part of a quantum-end event (see Chapter *Scheduling*)
- The Get Job/Process Information ($GETJPI) system service

These and other examples are described in Sections 8.8 and 8.9.

The third way to create an AST is an explicit declaration of an AST by a process through the $DCLAST system service.

The $DCLAST system service procedure, EXE$DCLAST in module SYSASTCON, runs in kernel mode. It simply allocates an ACB, initializes it with information from the system service arguments, and invokes SCH$QAST to queue the ACB. The access mode in which the AST is to execute can be no more privileged than the mode from which $DCLAST was requested. The system service charges the process AST quota.

8.6 Queuing an AST to a Process

The routine SCH$QAST, in module ASTDEL, queues an AST to a process. It can be invoked from any kernel mode thread of execution running at an IPL less than or equal to IPL$_SCHED and holding no spinlock of rank greater than SCHED. Basically, SCH$QAST inserts the ACB into one of the queues in the process's PCB.

SCH$QAST performs the following steps:

1. It acquires the SCHED spinlock, raising IPL to IPL$_SCHED, to synchronize access to the scheduler database and the process's AST-related data.
2. If the process is nonexistent, SCH$QAST releases the SCHED spinlock and returns the error status SS$_NONEXPR. If bit ACB$V_NODELETE in ACB$B_RMOD is clear, its usual state, SCH$QAST deallocates the ACB before returning.

   If the process exists, SCH$QAST gets the address of its software PCB.

3. It determines in which AST queue this ACB should go, based on ACB$V_KAST and the access mode bits in ACB$B_RMOD, and inserts the ACB at the tail of that queue:
   - — It inserts a special kernel mode AST in the special kernel mode AST queue.
   - — It inserts a normal AST in the queue corresponding to the mode bits.
   - — It inserts a piggyback special kernel mode AST in the queue corresponding to the mode of the normal AST whose ACB it shares.
4. It sets the bit corresponding to the AST mode in PCB$L_AST_PENDING.
5. It determines new contents for the ASTSR processor register or PHD$Q_ASTSR_ASTEN as a function of the old contents plus a set bit that corresponds to the mode of the newly queued ACB. It stores the contents as follows:
   - — If the target process is current, SCH$QAST stores the new ASTSR value in the processor register but not in PHD$Q_ASTSR_ASTEN. (While a process is current, software may not alter its HWPCB.)
   - — If the process is not currently executing, but its PHD is memory-resident, SCH$QAST stores the new value for ASTSR in PHD$Q_ASTSR_ASTEN but not in the processor register, which represents some other process's state. In addition, if delivery to that mode is enabled, SCH$QAST sets bit PHD$V_AST_PENDING in PHD$L_FLAGS2. Sometime later, when the process becomes current, if the flag is set, scheduling code may dispatch directly to an AST delivery interrupt service routine (see Section 8.7).
   - — If the process and its PHD are outswapped, PHD$Q_ASTSR_ASTEN cannot be updated because the PHD is not available. When the process later becomes resident, the swapper reinitializes the summary bits in PHD$Q_ASTSR_ASTEN, based on the state of the process's AST queues.

   When queuing a special kernel mode AST, SCH$QAST tests whether the process has disabled delivery of normal kernel mode ASTs. If so, SCH$QAST sets the low-order bit of PCB$L_AST_BLOCKED and reenables delivery to kernel mode to ensure that the special kernel mode AST will be delivered.

   SCH$QAST does not check whether delivery to the mode of the AST is disabled or whether there is an AST already active at that mode. If AST delivery is disabled for that mode, no AST interrupt will be requested. It is thus not necessary for SCH$QAST to make that check. If an AST is already active at that mode, an unnecessary AST interrupt for delivery to that mode will be requested, but the AST delivery interrupt service routine will dismiss it as blocked and undeliverable.
6. Unless the process is currently executing, SCH$QAST invokes SCH$RSE, in module RSE, to report that an AST has been queued to the process. SCH$RSE makes the process computable if it is not current, already computable, or suspended in kernel mode.
7. SCH$QAST releases the SCHED spinlock, restoring the previous IPL, and returns to its invoker.

8.7 Delivering an AST

AST delivery is initiated when a pending AST is deliverable and IPL is less than 2. For a pending AST to be deliverable, AST delivery to the access mode of the AST must be enabled and the access mode to be interrupted (or the mode being restored by the REI PALcode routine) must be less than or equally privileged to that of the AST. This test prevents a process running in an inner mode from being interrupted to deliver an AST to an outer mode.

AST delivery is initiated indirectly through one of the AST delivery interrupts. It can also be initiated directly by the rescheduling interrupt service routine, SCH$INTERRUPT, in module SCHEDULER.

After swapping process context, but before returning to the interrupted thread of execution in the newly current process, SCH$INTERRUPT tests the process's PHD$V_AST_PENDING bit. As previously described, SCH$QAST sets this bit when it queues an AST for a mode to which delivery is enabled. If the bit is set, SCH$INTERRUPT clears it, releases the SCHED spinlock it acquired at entry, and makes the following tests to determine whether an AST interrupt would be appropriate:

- It checks that the IPL to be restored is less than 2.
- It checks that the mode of the most privileged pending AST is equally or more privileged than the mode being restored.
- It checks that delivery to that mode is enabled.

If the process's AST state is such that an AST interrupt would be requested if SCH$INTERRUPT executed a `CALL_PAL REI` instruction to resume the interrupted thread, SCH$INTERRUPT itself initiates AST delivery. It clears the ASTSR bit corresponding to that mode, lowers IPL to 2, clears the software bits in the PS, and jumps directly to the appropriate AST interrupt service routine just as though an interrupt had occurred. This shortcut improves performance in cases where a waiting process is made computable by the queuing of a deliverable AST and then selected for execution. Because PHD$V_AST_PENDING is in a PHD field the scheduler accesses for other reasons, the cost of testing the bit is minimal.

8.7.1 AST Delivery Interrupt Service Routines

As described in Chapter *Interrupts and Exceptions,* there is a separate AST delivery interrupt for each access mode. When an AST delivery interrupt request is triggered and granted, the PALcode routine that initiates an exception or interrupt (referred to in this book as the IEI PALcode) executes. The routine transfers control through the appropriate system control block (SCB) entry to an interrupt service routine. The functions of each interrupt service routine are to remove the first pending AST from the queue associated with that mode, determine that the interrupt is not a spurious one, and dispatch to the specified AST routine at the specified access mode.

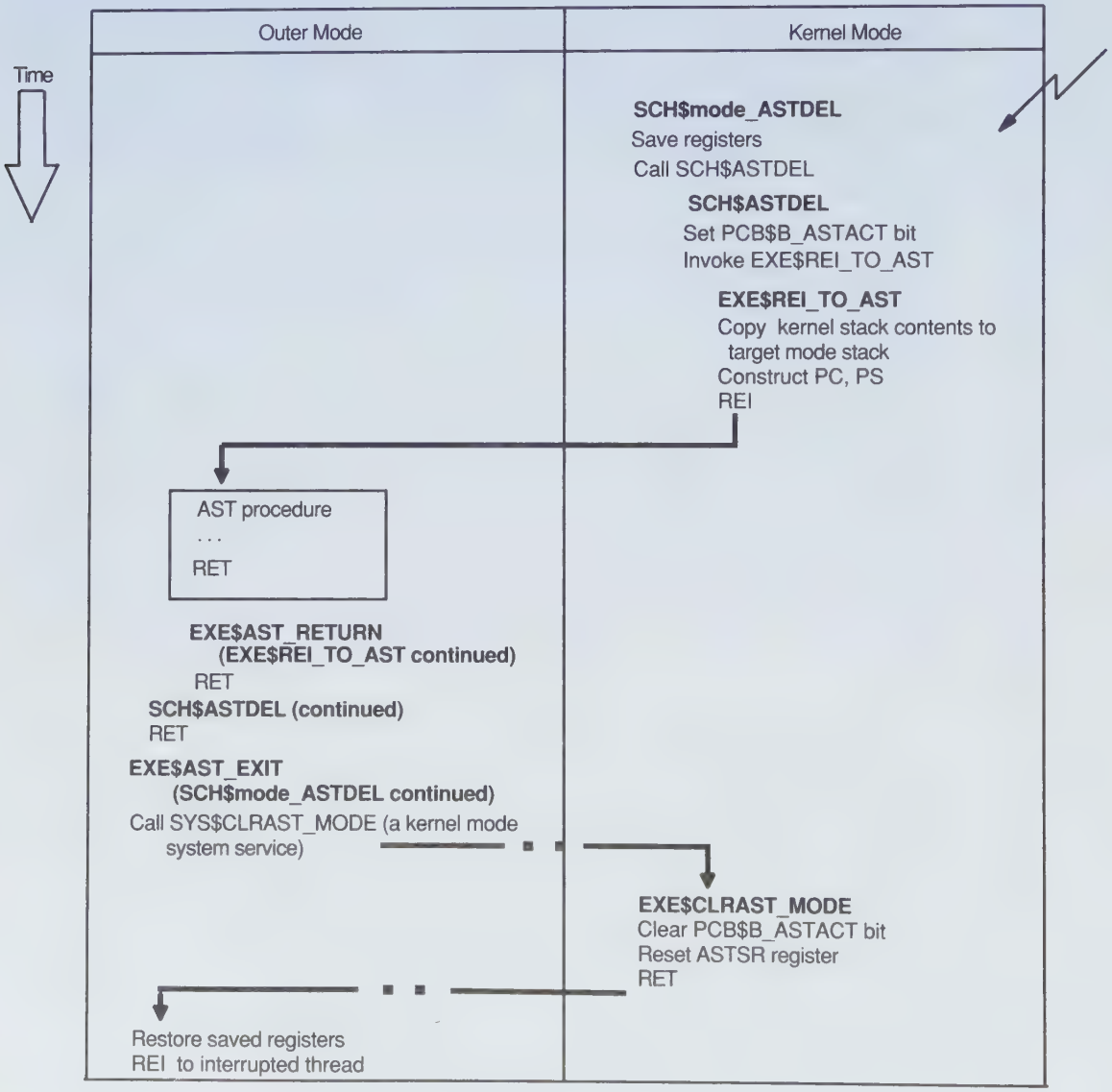
[*Censored*] VMS implements these interrupt service routines in a combination of VAX MACRO and [*censored*] assembly language. Stack manipulation necessary for AST delivery and exit must be written in [*censored*] assembly language.

Figure 8–3 shows the major steps in delivering an AST to an outer mode.

The *censored* assembly language AST delivery interrupt service routine entry points, in module ASTDEL_STACK, are

- SCH$USER_ASTDEL, for user mode ASTs
- SCH$SUPER_ASTDEL, for supervisor mode ASTs
- SCH$EXEC_ASTDEL, for executive mode ASTs
- SCH$KERNEL_ASTDEL, for both normal and special kernel mode ASTs

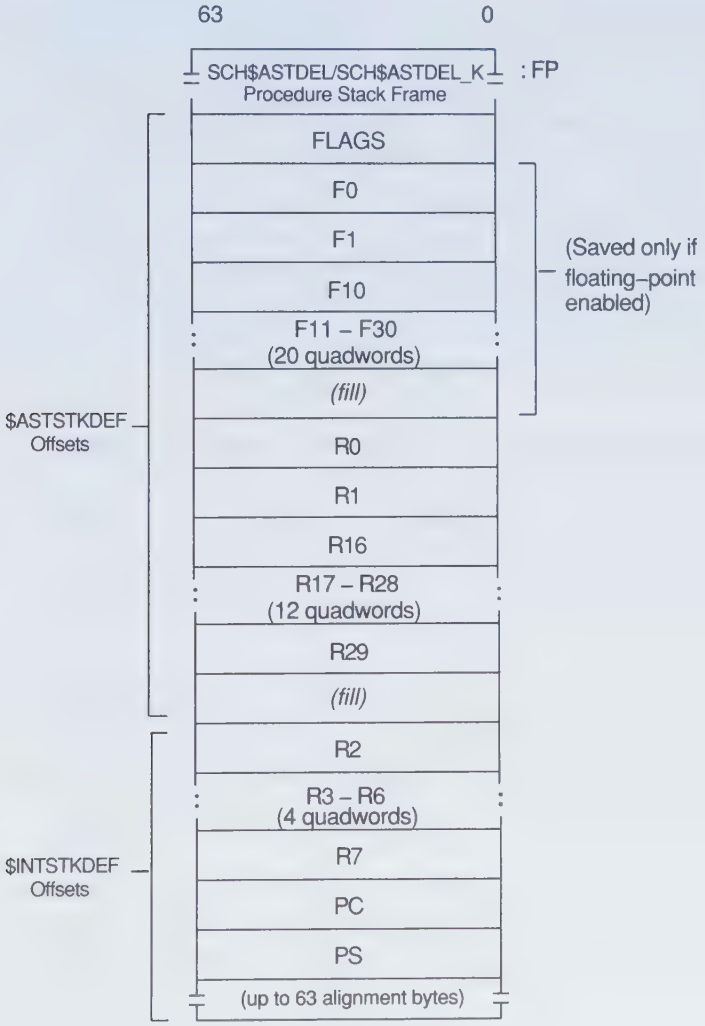
Figure 8–3 Outer Mode AST Delivery Flow



Each is entered in kernel mode at IPL 2 with the bit in ASTSR that corresponds to the AST mode already cleared. Each stores the AST access mode in a register and creates a stack save area in which it saves the integer registers that might otherwise be overwritten as a result of the AST procedure call or the service routine's actions. (The registers that compiler-generated code would not save and restore include the function value, argument passing, and scratch registers.) The macro $ASTSTKDEF defines symbolic offsets for this stack save area.

Figure 8–4 shows the contents of the stack during AST delivery: the interrupt stack frame, the stack save area, and the procedure stack frame built during a later step. Initially, all this information is on the kernel stack. Later it is copied to the stack of the AST access mode. Dispatch into the AST procedure adds another procedure stack frame.

Figure 8–4 AST Stack Contents



These interrupt service routines all converge in common code, but the path for kernel mode AST delivery differs in a number of ways, one of which is the delivery of special kernel mode ASTs. This section first describes outer mode delivery and then the differences in kernel mode delivery.

8.7.1.1 Outer Mode AST Delivery

The key concerns in delivering an AST to an outer mode are as follows:

- Making a mode transition from kernel to the outer access mode to call the AST procedure
- Returning from the AST procedure to the thread of execution interrupted by the AST interrupt with all its state restored

After creating the stack save area, each of the outer mode service routines tests the low bit of PHD$L_FLAGS (the software copy of the floating-point enable, FEN, register) to determine whether the process is using floating-point. If so, the scratch floating-point registers must be saved. (A *censored* processor has both a set of integer registers for addressing and integer arithmetic and a set of floating-point registers. Compiler-generated code for a routine that executes floating-point instructions is responsible for saving and restoring floating-point registers used by the routine, other than the scratch registers.)

If floating-point is not enabled for this process, each routine clears the stack quadword that represents the floating-point state. If it is enabled, each sets two flags in the quadword: ASTSTK$V_FEN, to indicate that floating-point is enabled, and ASTSTK$V_FP_SAVE_DELAYED, to indicate that the scratch floating-point registers have not yet been saved. (If they were saved now, they would have to be copied to the AST access mode stack later; instead, because the AST delivery code does not use them, they can be saved later.)

Each then calls SCH$ASTDEL, in module ASTDEL, specifying a return address of EXE$AST_EXIT.

SCH$ASTDEL, in combination with the *censored* assembly language routine EXE$REI_TO_AST, in module ASTDEL_STACK, takes the following steps:

1. SCH$ASTDEL acquires the SCHED spinlock, raising IPL to IPL$_SCHED, to synchronize access to the process's AST queues.
2. If an AST of that mode is already active (the corresponding PCB$B_ASTACT bit is set), it releases the SCHED spinlock and returns to EXE$AST_EXIT with a status indicating no AST was delivered. (SCH$ASTDEL leaves the access mode bit in ASTSR clear to prevent further spurious interrupts. When the AST procedure exits, the ASTSR bit will be reset.) AST exit code is described further in Section 8.7.3.

   If no AST of that mode is active, SCH$ASTDEL removes the first ACB from the appropriate PCB queue. If the queue was empty or is empty after the removal, SCH$ASTDEL clears the bit in PCB$L_AST_PENDING corresponding to the mode of the interrupt.
3. If the queue was empty, SCH$ASTDEL releases the SCHED spinlock, updates ASTSR, and returns to EXE$AST_EXIT with a status indicating no AST was delivered.

4. Otherwise, SCH$ASTDEL sets the bit corresponding to the AST access mode in PCB$B_ASTACT to indicate that there is an active AST at this mode and to block concurrent delivery of another AST.
5. If ACB$V_QUOTA is set in the ACB, SCH$ASTDEL returns process AST quota.
6. It releases the SCHED spinlock, lowering IPL to 2.
7. It copies the AST parameter from the ACB for use as an argument to the AST procedure.
8. If a piggyback special kernel mode AST is associated with this AST, SCH$ASTDEL tests whether the process is using floating-point. If so, it saves the scratch floating-point registers on the stack in case the piggyback kernel mode AST routine uses them and clears the ASTSTK$V_FP_SAVE_DELAYED bit.

   SCH$ASTDEL invokes the piggyback special kernel mode routine at IPL 2. When it returns, SCH$ASTDEL continues with step 10.
9. If the ACB$V_NODELETE bit is clear (its usual state), SCH$ASTDEL deallocates the ACB to nonpaged pool.
10. It initializes the other argument registers (see Section 8.7.2).
11. The code that actually calls an AST procedure must execute in the access mode of the AST. Before leaving kernel mode, however, AST delivery code must copy the kernel stack contents to the stack of the AST access mode so that the information is accessible from the outer mode at AST exit. SCH$ASTDEL invokes EXE$REI_TO_AST to perform that step and the rest of AST delivery.
12. EXE$REI_TO_AST first checks the accessibility of the AST procedure descriptor. If it cannot be read, EXE$REI_TO_AST returns an error status to SCH$ASTDEL.

    In response to that error status, SCH$ASTDEL clears the PCB$B_ASTACT bit, lowers IPL to 0, and returns to EXE$AST_EXIT with information about the failure. As described in Section 8.7.3, EXE$AST_EXIT signals the error SS$_ASTFLT.
13. If the AST procedure descriptor can be read, EXE$REI_TO_AST aligns the target access mode stack to a 64-byte boundary in preparation for copying the kernel stack contents, which include the interrupt stack frame. Later, after the AST procedure exits, control will be returned to the interrupted thread of execution through a `CALL_PAL REI` instruction, which requires the stack to be aligned.
14. EXE$REI_TO_AST probes the target mode stack to confirm that the contents of the kernel stack can be copied to it. If not, it returns an error status to SCH$ASTDEL. As described in Section 8.7.3, if the target mode stack cannot be expanded, SCH$ASTDEL clears the PCB$B_ASTACT bit, lowers IPL to 0, and returns to EXE$AST_EXIT. EXE$AST_EXIT signals the error SS$_ASTFLT.

15. If the target stack is accessible, EXE$REI_TO_AST modifies the target mode stack pointer to reflect its position after the copy and updates the frame pointer (FP) register to point to the location on the target mode stack of the moved stack frame from the entry to SCH$ASTDEL.

    It copies the information pushed onto the kernel stack by SCH$ASTDEL's prologue code, namely, SCH$ASTDEL's procedure stack frame.

    It tests ASTSTK$V_FEN to determine whether the process is using floating-point. If not, it continues with the next step. If so, it tests whether the scratch floating-point registers have been saved already, perhaps prior to the execution of a piggyback special kernel mode AST routine. If they have, EXE$REI_TO_AST reloads them from the kernel stack save area and clears its ASTSTK$V_FP_SAVE_DELAYED flag. It stores the scratch floating-point registers on the target access mode stack.

16. It copies the kernel stack save area and the interrupt stack frame to the target stack. It adjusts the stack alignment bits in the saved processor status (PS) in the copied interrupt stack frame to reflect the target mode stack alignment described in step 13.

17. It modifies the interrupt stack frame on the kernel stack so that it can REI to the target mode and enter the AST procedure: it inserts the target mode as previous and current in the PS to be restored; it saves its own caller's return address (in SCH$ASTDEL) in the R2 contents to be restored from the interrupt stack frame and loads R26 with the address of EXE$AST_RETURN, a return address within EXE$REI_TO_AST.

18. EXE$REI_TO_AST examines the AST's procedure descriptor to determine whether it is a native procedure or a translated procedure.

    If it is a native procedure, EXE$REI_TO_AST stores its entry point address in the program counter (PC) to be restored, loads its procedure value into R27, and executes a `CALL_PAL REI` instruction, entering the AST procedure in its access mode.

    If the AST procedure is translated, EXE$REI_TO_AST instead stores the entry point address of SYS$NATIVE_TO_TRANSLATED and loads its procedure value. The jacket routine SYS$NATIVE_TO_TRANSLATED will call the AST procedure itself. Chapter *Translated Image Environment* provides more information on the translated image environment.

8.7.1.2 Kernel Mode AST Delivery

The kernel mode AST delivery interrupt service routine is concerned with

- Delivering all queued special kernel mode ASTs in response to one interrupt
- Delivering one normal AST as well, if possible

SCH$KERNEL_ASTDEL performs the same actions as the other AST delivery interrupt service routines except that, if floating-point is enabled, it saves the floating-point registers, since the current stack is the AST access mode stack. It then calls SCH$ASTDEL_K, in module ASTDEL, specifying a return address of EXE$AST_EXIT.

SCH$ASTDEL_K takes the following steps:

1. It checks for the relatively common case of a special kernel mode AST queue that contains exactly one ACB. If the queue contains exactly one ACB, SCH$ASTDEL_K removes it and updates the listhead using the load-locked/store-conditional mechanism, described in Chapter *Synchronization Techniques*, without the overhead of acquiring and releasing the SCHED spinlock. It continues with step 5 to deliver the special kernel mode AST.

   If the queue is empty or contains more than one ACB, SCH$ASTDEL_K continues with the next step.

2. It acquires the SCHED spinlock, raising IPL to IPL$_SCHED.

3. It removes the ACB from the head of the special kernel mode AST queue. If the queue is empty, it continues with step 7.

4. It releases the SCHED spinlock, lowering IPL to 2.

5. It clears the ACB$V_KAST bit in ACB$B_RMOD.

6. It invokes the special kernel mode AST routine, passing it the addresses of the PCB and ACB in R4 and R5.

   When the special kernel mode AST routine returns, SCH$ASTDEL_K continues with step 2, delivering all pending special kernel mode ASTs.

7. When the special kernel mode AST queue is empty, SCH$ASTDEL_K tests and clears the low bit of PCB$L_AST_BLOCKED.

8. If the bit was set, indicating that kernel mode AST delivery should be disabled, SCH$ASTDEL_K disables kernel mode delivery by executing a `CALL_PAL SWASTEN`. It releases the SCHED spinlock, updates the ASTSR register with the contents of PCB$L_AST_PENDING, and returns to EXE$AST_EXIT with a status indicating no normal AST was delivered.

9. If the low bit of PCB$L_AST_BLOCKED was clear, SCH$ASTDEL_K tests the kernel mode PCB$B_ASTACT bit. If the bit is set, indicating a kernel mode AST already active, it releases the SCHED spinlock and returns to EXE$AST_EXIT with a status indicating no normal AST was delivered.

10. If the PCB$B_ASTACT bit was clear, SCH$ASTDEL_K removes the first ACB from the kernel mode PCB queue. If the queue was or is now empty, it clears the kernel mode bit in PCB$L_AST_PENDING. If SCH$ASTDEL_K did remove an ACB, its flow converges with that of SCH$ASTDEL, at step 4. (Note, however, that because no access mode change is necessary, the stack is already set up for calling the AST procedure; much of the stack manipulation in EXE$REI_TO_AST is bypassed.)

11. If the queue was empty, SCH$ASTDEL_K examines the pending ASTs to determine whether there is one whose mode is enabled and whose mode is equal to or more privileged than that of the interrupted thread.

    If there is none, SCH$ASTDEL_K releases the SCHED spinlock and returns to EXE$AST_EXIT with a status indicating no normal AST was delivered.

12. If there is an appropriate pending AST, SCH$ASTDEL_K updates the ASTSR register by clearing the bit corresponding to the mode of the AST, as PALcode or hardware-initiated delivery would have done, and merges with SCH$ASTDEL at step 2.

8.7.2 AST Arguments

An AST procedure is called with the following arguments:

- AST parameter
- Contents of R0 at the time of the interrupt
- Contents of R1 at the time of the interrupt
- The PC of the interrupted thread of execution
- The PS of that thread

The only argument directly intended for the AST procedure is the AST parameter, which was originally an argument to a system service such as $QIO, $ENQ, or $DCLAST. SCH$ASTDEL copies the AST parameter from the ACB where it was initially stored by the system service. Its meaning is application-specific.

Although the other arguments are present for compatibility with VAX/VMS, they have no subsequent use after the AST procedure exits; modifying them therefore has no effect on the thread of execution to be resumed at AST exit.

8.7.3 AST Exit Path

After an AST procedure has executed, its associated PCB$B_ASTACT bit must be cleared and ASTSR must be recomputed. Requiring kernel mode, these steps are performed by the $CLRAST and $CLRAST_MODE system services. (The distinction between the two forms of this service is that the latter is passed an argument containing the access mode.) In most cases, the AST procedure implicitly requests the $CLRAST_MODE system service by simply returning. Direct request of these system services is discussed later in this section.

When the AST procedure returns, the following steps occur:

1. By returning, the AST procedure transfers control to EXE$AST_RETURN, in module ASTDEL_STACK, which simply loads a success status into R0 and returns to SCH$ASTDEL.
2. SCH$ASTDEL tests R0 to distinguish successful AST execution from bad stack and inaccessible procedure descriptor errors.
3. If R0 contains a success status, SCH$ASTDEL continues with the actions in step 7.
4. If R0 contains an error status, SCH$ASTDEL, still in kernel mode (because no AST was delivered), determines which error occurred. If the target access mode stack could not accommodate the kernel stack contents, SCH$ASTDEL tests whether the target mode is user. If so, it invokes EXE$EXPANDSTK, in module EXCEPTION, to expand the user stack. Chapter *Memory Management System Services* contains further details. If the expansion succeeds,

SCH$ASTDEL invokes EXE$REI_TO_AST again to deliver the AST (see step 11 and subsequent steps in the description of SCH$ASTDEL in Section 8.7.1.1).

If the expansion fails or the target mode is not user, SCH$ASTDEL loads registers with information about the error and continues with step 6.

5. If the AST procedure descriptor is inaccessible, SCH$ASTDEL loads registers with information about the error.

6. It clears the PCB$B_ASTACT bit for the AST mode, lowers IPL to 0, and returns to EXE$AST_EXIT, in module ASTDEL_STACK, with an error status. Returning, SCH$ASTDEL's epilogue code removes the procedure stack frame and restores the registers saved by SCH$ASTDEL's prologue code.

   EXE$AST_EXIT restores the registers from the stack save area and removes the stack save area from the stack, leaving only the interrupt stack frame. It jumps to EXE$ASTFLT. Chapter *Condition Handling* describes EXE$ASTFLT's actions to signal the error.

7. If R0 contains a success status, the AST executed successfully. Running in the access mode of the AST, SCH$ASTDEL returns to its caller, the AST delivery interrupt service routine, at EXE$AST_EXIT. Returning, SCH$ASTDEL's epilogue code removes the procedure stack frame and restores the registers saved by SCH$ASTDEL's prologue code.

8. EXE$AST_EXIT, still running in the mode of the AST, temporarily disables AST delivery to that mode to block a possible AST interrupt until step 10b. It tests to determine in which access mode it is. If it is in an outer mode, EXE$AST_EXIT requests the $CLRAST_MODE system service. If already in kernel mode, EXE$AST_EXIT simply invokes EXE$CLRAST_KERNEL, an alternative entry point of EXE$CLRAST, bypassing the unnecessary system service dispatch.

9. The $CLRAST and $CLRAST_MODE system service procedures, EXE$CLRAST and EXE$CLRAST_MODE in module ASTDEL, perform the following steps:

   a. EXE$CLRAST extracts the previous mode bits from the PS to determine the relevant access mode; EXE$CLRAST_MODE uses the mode passed to it.

   b. Each clears the corresponding PCB$B_ASTACT bit to indicate that no AST procedure is active in that mode.

   c. Each invokes SCH$RESET_ASTSR, in module ASTDEL, to update the ASTSR register.

   d. Each returns.

10. EXE$AST_EXIT resumes at the previous access mode, the mode of the AST. (If EXE$AST_EXIT had not disabled AST delivery, the REI from the system service dispatch return could have triggered an AST interrupt for a pending AST at this access mode.)

    a. It restores the general registers and, if saved, the floating-point registers from the stack save area.

b. It restores the previous state of the AST enable for this mode by executing a CALL_PAL SWASTEN instruction. Typically, this reenables delivery to the current mode, making delivery of another AST to this mode possible.

c. EXE$AST_EXIT executes a CALL_PAL REI instruction to dismiss the interrupt and remove the interrupt stack frame from the stack. This returns control to the access mode and instruction originally interrupted by AST delivery.

The CALL_PAL SWASTEN instruction in EXE$AST_EXIT, at step 10b, can cause another AST delivery interrupt to occur, depending upon the ASTSR contents and the access mode transitions.

If another AST delivery interrupt does occur before the CALL_PAL REI in step 10c dismisses the current one, the interrupt stack frame of the nearly completed current AST is still on the stack. Allowing subsequent AST interrupts to nest within the current one could potentially fill the stack with interrupt stack frames. SCH$ASTDEL attempts to avoid both the nesting and the CALL_PAL REI. It checks whether an AST interrupt occurred at the CALL_PAL SWASTEN instruction by examining the PC in the newer interrupt stack frame. If it did, SCH$ASTDEL checks further whether the current AST and the previous AST are for the same access mode. If they are, SCH$ASTDEL invokes EXE$REI_TO_AST with a flag indicating that the newer interrupt stack frame should not be saved. For an outer mode AST, this means it is not copied to the target stack; for a kernel mode AST, it means that the ASTSTK save area and SCH$ASTDEL procedure stack frame overlay the newer interrupt stack frame.

If an AST procedure requests the $CLRAST or $CLRAST_MODE system service directly rather than simply returning, the appropriate PCB$B_ASTACT bit is cleared and ASTSR register is updated. This has the effect that another AST can be delivered to the same mode; the current procedure is now an ordinary thread interruptible by ASTs. The procedure stack frame built by the call to SCH$ASTDEL remains on the stack, as do the stack save area and interrupt stack frame. The former AST procedure must return in order to remove this information from the stack and restore saved registers before returning to the code interrupted by the AST delivery. Furthermore, the former AST procedure is now responsible for any synchronization with another AST thread of execution.

The VAX BASIC Run-Time Library can request the $CLRAST system service from within CTRL/C attention AST procedures. VAX BASIC requires that user programs be notified of CTRL/C through an error signal rather than through the AST mechanism. If no I/O request is in progress, the VAX BASIC Run-Time Library dismisses the CTRL/C attention AST by requesting the $CLRAST system service and then signals the condition by calling LIB$SIGNAL (see Chapter *Condition Handling*). If an I/O request is currently in progress, the VAX BASIC Run-Time Library sets an internal flag, returns from the AST procedure, and later, when the I/O request completes, signals the CTRL/C error.

Note that the $CLRAST and $CLRAST_MODE system services are not supported by Digital, except for use within Digital software, and are not documented in the *VMS System Services Reference Manual*.

8.8 Special Kernel Mode ASTs

Special kernel mode ASTs differ from normal ASTs in several ways:

- A special kernel mode AST routine is dispatched at IPL 2 and executes at that level or higher. Synchronization is provided by the interrupt mechanism itself rather than requiring an additional PCB$B_ASTACT bit. Only one special kernel mode AST can be active at any time because the kernel mode AST delivery interrupt is blocked.
- Special kernel mode ASTs cannot be disabled by clearing the kernel mode ASTEN bit. Delivery of a special kernel mode AST can only be blocked by raising IPL to 2 or above.
- All special kernel mode ASTs result from the operations of kernel mode code. That is, a user cannot directly request special kernel mode AST notification of an asynchronous event.
- A special kernel mode AST routine is identified as a `.JSB_ENTRY` entry point rather than as a `.CALL_ENTRY` entry point.

  The arguments passed to a special kernel mode AST routine are the PCB address in R4 and the ACB address in R5. When the routine exits, the stack must be in the same state as when the routine was entered. The routine may use R0 through R5 but must save other registers defined to be nonscratch by the *censored* Calling Standard before use and restore them before exiting. A special kernel mode AST routine written in a language other than *censored* assembly language leaves this detail to the compiler.
- As the result of one interrupt, all pending special kernel mode ASTs are delivered and possibly one other pending deliverable AST.
- A special kernel mode AST routine is responsible for the deallocation of the ACB to nonpaged pool. For normal ASTs, this deallocation is done by the AST delivery routine.

The next several sections briefly describe examples of the special kernel mode AST mechanism.

8.8.1 I/O Postprocessing in Process Context

Completing an I/O request requires the delivery of a special kernel mode AST to the process whose I/O completed. I/O postprocessing is described in more detail in Chapter *I/O Processing*.

The I/O postprocessing interrupt service routine queues a former IRP as an ACB to the process whose I/O completed. The operations performed by the I/O completion AST routine are those that must execute in process context, particularly those that reference process virtual addresses.

The special kernel mode AST routines BUFPOST and DIRPOST, in module IOCIOPOST, perform the following operations (DIRPOST is actually a subentry point of BUFPOST):

1. For buffered read I/O operations only, BUFPOST copies the data from the system buffer to the user buffer in process address space and deallocates the system buffer to nonpaged pool.
2. If a user diagnostic buffer was associated with the I/O request, DIRPOST copies the diagnostic information from the system diagnostic buffer to the user's buffer and deallocates the system buffer.
3. DIRPOST decrements the channel control block field CCB$W_IOC, the number of I/O requests in progress on this channel. Channel control blocks are in P1 space.
4. If the I/O request specified an I/O status block (IOSB), the routine copies information from the IRP to the IOSB, which is in process address space.
5. If a common event flag is associated with the I/O request, it is set. Chapter *I/O System Services* explains why common event flags must be set in the special kernel mode AST routine though local flags are set in IOC$IOPOST.
6. If ACB$V_QUOTA was set in IRP$B_RMOD (the same offset as ACB$B_RMOD), AST notification of I/O completion was requested. The AST procedure value and the optional AST argument were originally stored in the IRP (now an ACB). DIRPOST invokes SCH$QAST to requeue the former IRP as an ACB. This time the IRP/ACB represents a normal AST in the access mode at which the I/O request was made.
7. Otherwise, if ACB$V_QUOTA is clear, DIRPOST deallocates the IRP/ACB to nonpaged pool.

8.8.2 $GETJPI System Service

A process requests the $GETJPI system service to obtain information about itself or another process. If the request is for information in the virtual address space of another process, the $GETJPI system service queues an AST to the target process. Running in the context of the target process, $GETJPI's special kernel mode AST routine can easily examine per-process address space. Chapter *Process Control and Communication* describes the $GETJPI system service in detail.

The $GETJPI system service procedure, EXE$GETJPI in module SYSGETJPI, performs the following steps:

1. It allocates and fills in an extended ACB to describe a special kernel mode AST and the desired items of information. The ACB includes a buffer to return the data.
2. The special kernel mode AST routine, executing in the context of the target process, moves the requested information into the buffer. It modifies the ACB so that it can be used to queue a second special kernel mode AST back to the requesting process.
3. The second special kernel mode AST routine copies data from the extended ACB buffer to buffers in the requesting process. It also sets the event flag associated with this request.

4. If the process has requested AST notification of request completion, the extended ACB is used for the third time. The special kernel mode AST routine uses it to cause delivery of a normal AST in the access mode from which the system service was requested.

   If the process has not requested AST notification, the extended ACB is deallocated to nonpaged pool.

8.8.3 Power Recovery ASTs

The implementation of power recovery ASTs relies on special kernel mode ASTs. A power recovery AST enables a process to receive notification that a power failure and successful restart have occurred. Chapter *Power Failure and Recovery* describes this feature in more detail.

When a power recovery occurs, VMS queues a special kernel mode AST to each process that has requested power recovery AST notification. The special kernel mode AST routine copies the procedure value of the user-requested AST procedure, which is stored in P1 space, to ACB$L_AST and requeues the ACB as a normal AST. The special kernel mode AST routine is necessary for accessing the process's P1 space.

8.8.4 Piggyback Special Kernel Mode ASTs

Piggyback special kernel mode ASTs (PKASTs) enable a special kernel mode AST to ride piggyback in the ACB$L_KAST field of a normal AST. The normal access mode determines the order of enqueuing and delivery. If delivery to that access mode is disabled or blocked, the piggyback special kernel mode AST cannot be delivered.

The AST delivery interrupt service routine invokes the piggyback special kernel mode AST routine just before calling the normal AST. When the special kernel mode AST returns, the normal AST is called.

There are several reasons for using piggyback special kernel mode ASTs:

- It is faster to deliver two ASTs together than to deliver two ASTs separately.
- There are times when delivering an AST requires some additional work in kernel mode in the context of the target process. Piggyback special kernel mode ASTs facilitate this work.

  The $ENQ system service uses a piggyback special kernel mode AST to write to the requestor's lock status block and lock value block. To copy the information from the lock database to the requestor's process space, a piggyback special kernel mode AST is required.

  Piggyback special kernel mode ASTs are also used in terminal out-of-band ASTs (see Section 8.10.5.3).
- A piggyback special kernel mode AST can queue other normal ASTs to a process. The $ENQ system service uses this feature to deliver both blocking and completion ASTs to a process through one ACB. Chapter *Lock Management* contains further information.

8.9 System Use of Normal ASTs

Several other executive features are implemented through normal ASTs. For example, the automatic working set limit adjustment that takes place at quantum end is implemented with a normal kernel mode AST. Chapter *Scheduling* discusses quantum-end activities, and Chapter *Working Set List Dynamics* provides a detailed description of automatic working set limit adjustment.

CPU time limit expiration is implemented with potentially multiple ASTs. Beginning in user mode, the AST procedure requests the Exit ($EXIT) system service. If the process is not deleted, a supervisor mode time expiration AST is queued. This loop continues with higher access modes until the process is deleted.

The executive also uses the AST mechanism for the $FORCEX, Suspend Process ($SUSPND), and Delete Process ($DELPRC) system services. Each system service procedure queues an AST to the target process.

The $FORCEX system service, detailed in Chapter *Process Control and Communication*, queues a user mode AST that requests the $EXIT system service from the context of the target process.

The $SUSPND and $DELPRC system services queue an AST to the target process to implement suspension or deletion through code running in the context of the target process.

The $SUSPND system service queues either a supervisor or kernel mode AST to its target process, depending on the access mode of the suspension. A process suspended through a supervisor mode AST (the default) can execute kernel and executive mode ASTs. Supervisor mode suspension is described in greater detail in Chapter *Process Control and Communication*. A process suspended through a kernel mode AST can become computable only when it is resumed through another process.

Process deletion and kernel mode suspension must take care to synchronize their actions with the activities of the Files-11 XQP.

The Files-11 XQP runs in process context as a kernel mode AST thread, taking out locks and making I/O requests in response to the process's file system requests. The XQP indicates that it is active by incrementing the PCB field PCB$B_DPC. When the XQP must wait for a lock to be granted or an I/O request to complete, it returns from the AST procedure so that the process can wait at the access mode in which the file system request originated.

Waiting in the outer mode allows delivery of ASTs to that mode and more privileged modes. While the XQP is executing or waiting, kernel mode suspension of the process would risk blocking other processes with interests in the same locks. Deletion of the process would risk relatively minor on-disk corruption, such as dangling directory entries and lost files.

Therefore, the kernel mode suspension and process deletion services queue normal kernel mode ASTs, which cannot be delivered until the XQP AST completes. Furthermore, these AST procedures check that PCB$B_DPC is zero before proceeding with actual process suspension or deletion.

If PCB$B_DPC is not zero, these AST procedures place the process into a wait. They clear bit 0 of PCB$B_ASTACT so that another kernel mode AST can be delivered, invoke SCH$RESET_ASTSR to reset the ASTSR register, and place the process into the resource wait RSN$_ASTWAIT. The process waits in kernel mode at IPL 0. Thus, special and normal kernel mode ASTs can be delivered to it. The resource wait PC is an address within the AST procedure, so after the XQP AST completes, the suspend or delete AST procedure will be reentered to finish its job.

Some time later, queuing of an AST makes the process computable, and delivery of an XQP completion AST causes the XQP to be reentered. When the XQP is done, it decrements PCB$B_DPC and returns from its AST procedure. The suspend or delete AST procedure is reentered and can proceed, now that PCB$B_DPC is zero.

8.9.1 Process Suspension

The $SUSPND system service causes a target process to be placed into a suspended state. The system service procedure first checks the capability of the initiating process to affect the target process (see Chapter *Process Control and Communication* for further details). It then checks whether a supervisor or kernel mode suspension is requested. Supervisor mode is the default. A kernel mode suspension request, specified in the optional FLAGS argument, must be made from executive or kernel mode.

The system service procedure then sets PCB$V_SUSPEN in the target process's PCB$L_STS and, for a supervisor mode suspension, PCB$V_SOFTSUSP as well. It then queues either a kernel or supervisor mode AST to the target process so that the suspension and waiting will occur in that process's context. The wait mechanism in VMS requires that a process be placed into a wait from its own context.

When the kernel mode AST is delivered to a process undergoing kernel mode suspension, the SUSPND AST procedure acquires the SCHED spinlock, raising IPL to IPL$_SCHED, and tests whether PCB$V_RESPEN in PCB$L_STS is set. The bit, when set, indicates that a Resume Process ($RESUME) system service has been requested for this process. If the bit is set, the SUSPND AST procedure clears both it and PCB$V_SUSPEN and returns, leaving the process unsuspended.

If a $RESUME has not been requested for this process, SUSPND tests PCB$B_DPC to determine whether an XQP operation is in progress. If PCB$B_DPC is greater than zero, SUSPND places the process into a resource wait as previously described.

If PCB$B_DPC is zero, SUSPND places the process into a suspended wait state. The process waits in kernel mode at IPL 0. Its saved PC is an address within SUSPND, so when the process is later placed into execution, it again tests whether a $RESUME has been requested.

When the supervisor mode AST is delivered to a process undergoing supervisor mode suspension, the SUSPEND_SOFT AST procedure requests the $SUSPND system service. Running in kernel mode in the context of the target process, the $SUSPND system service procedure acquires the SCHED spinlock and tests whether PCB$V_RESPEN is set. If a $RESUME has not been requested for the

process, the $SUSPND system service procedure cleans up the kernel stack and places the process into a suspended wait state. These actions can only be done from kernel mode.

The process waits in supervisor mode with the supervisor mode PCB$B_ASTACT bit set. Its saved PC is an address within the SUSPEND_SOFT AST procedure, so when the process is placed back into execution, it again requests the $SUSPND system service to test whether a $RESUME has been requested. Waiting in this manner, the process can execute kernel and executive mode ASTs. For further details, see Chapter *Process Control and Communication*.

8.9.2 Process Deletion

The $DELPRC system service causes a target process to be deleted. After checking the capability of the initiating process to affect the target process (see Chapter *Process Control and Communication*), the system service procedure queues a normal kernel mode AST to the target process so that the deletion will occur in the context of that process. Chapter *Process Deletion* provides a detailed explanation of process deletion.

The use of the AST mechanism provides the following advantages:

- Queuing the AST makes the process computable, regardless of its wait state, unless the process is suspended. The $DELPRC system service ensures the deletion of a suspended process by requesting the $RESUME system service before queuing the AST.
- The process must be resident for the AST to be delivered. Therefore, special cases, such as the deletion of a process that is outswapped, simply do not exist.
- The DELETE AST procedure, running in process context, is able to request standard system services, such as Deassign Channel ($DASSGN), Deallocate Device ($DALLOC), and Delete Virtual Address Space ($DELTVA), to implement process deletion. These system services and the AST procedure reference per-process address space, and thus they must run in process context.

8.10 Attention and Out-of-Band ASTs

Several VMS device drivers queue an AST to notify a process that a particular attention condition has occurred on a device. The terminal driver, for example, queues an attention AST to notify an interested process that CTRL/C or CTRL/Y has been typed on its terminal. The terminal driver can also queue an out-of-band AST as notification that a control character other than CTRL/C and CTRL/Y has been typed. The mailbox driver can queue an attention AST as notification that an unsolicited message has been put in a mailbox or that an attempt to read an empty mailbox is in progress.

The basic sequence for both attention ASTs and out-of-band ASTs follows:

1. A process assigns a channel and requests the $QIO system service, specifying that it should receive AST notification of an attention condition on that device.

2. The device driver builds a data structure to describe the attention AST request, inserts it on a list connected to the device unit control block (UCB), and completes the I/O request.
3. If the attention condition occurs, the device interrupt service routine delivers the attention AST by queuing an AST to the process.

The major distinction between the attention AST and the out-of-band AST mechanisms is that out-of-band ASTs automatically repeat, whereas attention ASTs must be "rearmed." That is, a process must repeat its $QIO request for each attention notification.

Attention ASTs are described in the following sections, and out-of-band ASTs are described in Section 8.10.5.

8.10.1 Requesting an Attention AST

To establish an attention AST for a particular device whose driver supports this feature, the user requests the $QIO system service with the I/O function IO$_SETMODE or, for some devices, IO$_SETCHAR. The kind of attention AST requested is indicated by a function modifier.

The relevant function decision table (FDT) action routine for such a driver invokes COM$SETATTNAST, in module COMDRVSUB, which performs the following actions:

1. If the user AST procedure value (the $QIO P1 parameter) is zero, the request is interpreted as a flush attention AST list request (see Section 8.10.3).
2. Otherwise, COM$SETATTNAST allocates an expanded ACB from nonpaged pool and charges it against the process AST quota. The expanded ACB will be used both as a fork block (FKB) and as an ACB and is referred to as a FKB/ACB.
3. COM$SETATTNAST copies information into the FKB/ACB, such as the procedure value of the AST procedure, AST argument, channel number, and PID.
4. It acquires the device lock, raising IPL to UCB$B_DIPL, to synchronize access to the attention AST list. An attention AST list is a singly linked, last-in/first-out (LIFO) list of FKB/ACBs connected to the UCB of a device. COM$SETATTNAST inserts the FKB/ACB into the attention AST list.

   The location of the FKB/ACB listhead is driver-specific; some UCBs have multiple listheads, one for each attention condition the driver supports. The FDT action routine passes the address of the listhead in a register to COM$SETATTNAST.
5. COM$SETATTNAST then releases the device lock, restoring the previous IPL, and returns to the FDT action routine.

8.10.2 Queuing Attention ASTs

When the driver (typically the device interrupt service routine) determines that the attention condition has occurred, it invokes COM$DELATTNAST with the address of the FKB/ACB listhead.

COM$DELATTNAST queues every FKB/ACB on the list to its target process. A driver invokes an alternative entry point, COM$DELATTNASTP, to specify that only ASTs requested by a particular process be queued. (Each process that has assigned a channel to the device can request AST notification of an attention condition.)

COM$DELATTNAST is entered at device IPL with the device lock held to synchronize access to the attention AST list. The queuing of ASTs must occur at IPL$_SCHED with the SCHED spinlock held to synchronize access to the scheduler database (see Chapter *Synchronization Techniques*). Specifically, IPL must not be lowered to IPL$_SCHED. To accomplish correct synchronization and not block activities at IPL 7 and IPL 8, COM$DELATTNAST creates an IPL$_QUEUEAST (6) fork process to queue each AST.

The following steps summarize queuing of attention ASTs:

1. COM$DELATTNAST scans each FKB/ACB in the list.

   In the case of entry through COM$DELATTNASTP, the routine compares the PID in the FKB/ACB to the requested PID. If they are not equal, the routine leaves the data structure in the queue and goes on to the next entry. If the PIDs match, the routine continues with the next step.

2. The routine removes the FKB/ACB from its list, stores the procedure value of a fork routine in FKB$L_FPC of the FKB/ACB, and dispatches to EXE$PRIMITIVE_FORK. EXE$PRIMITIVE_FORK queues the fork block to the fork IPL 6 listhead and requests an interrupt at that IPL.

3. When IPL drops below 6, the fork interrupt is granted. The IPL 6 fork dispatcher removes the FKB/ACB from the IPL 6 fork block queue and dispatches to COM$DELATTNAST's fork process routine.

4. At IPL 6, COM$DELATTNAST's fork process routine reformats the fork block into an ACB, describing the AST procedure and the access mode of the original attention AST request.

5. The fork process routine invokes SCH$QAST, which acquires the SCHED spinlock and then queues the ACB to the process that requested the attention AST.

8.10.3 Flushing an Attention AST List

The list of attention ASTs is flushed as the result of an explicit user request, a Cancel I/O ($CANCEL), or a $DASSGN system service request for the associated device.

A user explicitly requests that the attention AST list be flushed by requesting the $QIO system service with an I/O function code and modifier for establishing an attention AST but with an AST procedure value of zero (see Section 8.10.1). Invoked this way, COM$SETATTNAST branches to COM$FLUSHATTNS.

COM$FLUSHATTNS is entered with the PID and channel number of the attention ASTs to be deleted. COM$FLUSHATTNS performs the following operations:

1. It acquires the device lock, raising IPL to UCB$B_DIPL of the device.
2. It scans the FKB/ACB list looking for all FKB/ACBs with a PID and channel number that match those of the requested flush operation.
3. For each FKB/ACB with matching PID and channel number, COM$FLUSHATTNS removes it from the attention AST list. It releases the device lock, restoring the IPL at which it was entered. It increments the process AST quota and deallocates the FKB/ACB to nonpaged pool.
4. It continues its scan of the list with step 1. Reaching the end of the list, COM$FLUSHATTNS releases the device lock and returns to its invoker.

8.10.4 Examples in the VMS Executive

Brief descriptions follow of the terminal driver's and mailbox driver's support of attention ASTs.

8.10.4.1 Terminal Driver and CTRL/C–CTRL/Y Notification

Notification of CTRL/C and CTRL/Y is requested through a $QIO to the terminal driver with a function code of IO$_SETMODE or IO$_SETCHAR and function modifier of IO$M_CTRLCAST or IO$M_CTRLYAST. There is a separate listhead for each condition in each terminal UCB.

When DCL is first entered in a process, it requests CTRL/Y notification; because it executes in supervisor mode, its attention AST will be a supervisor mode AST. A user image, running in the same process, can also request CTRL/C and CTRL/Y notification. Typically, the image's attention ASTs will be user mode ASTs. Note that DCL's supervisor mode AST would be entered first if CTRL/Y is typed. If an application is to receive control in response to CTRL/Y, DCL must cancel its request. Typically, such an application calls the Run-Time Library procedure LIB$DISABLE_CTRL; alternatively, the application could run in a process in which the DCL command SET NOCONTROL=Y had been issued.

When an interactive user (or the running image) spawns a subprocess, DCL, running in the context of the spawned subprocess, requests CTRL/Y notification. The image running in the spawned subprocess can also request CTRL/C and CTRL/Y notification. All the attention AST requests, because they are associated with the same terminal, are queued to the same CTRL/C and CTRL/Y listheads. As the user spawns a new subprocess or attaches to an already created process, DCL, running in the context of that process, tells the terminal driver the PID of the process currently associated with the terminal.

When CTRL/C is typed, the terminal driver invokes COM$DELATTNASTP to deliver only those CTRL/C attention ASTs requested by the process currently associated with the terminal. If no CTRL/C attention AST has been requested, the terminal driver interprets CTRL/C as CTRL/Y and searches the CTRL/Y AST list instead. When a CTRL/Y is typed, the driver searches only the CTRL/Y attention AST list. For more information on spawn and attach, see Chapter *Process Dynamics*.

Because the FKB/ACB data structures are not reused, CTRL/C and CTRL/Y attention ASTs must be reenabled each time they are delivered to a process.

The CTRL/Y attention AST list is flushed by a $DASSGN request. The CTRL/C attention AST list is flushed by $CANCEL as well as by $DASSGN. Both lists can be flushed by an explicit user request.

8.10.4.2 Mailbox Driver

A process requests a mailbox attention AST by requesting the $QIO system service with the function code IO$_SETMODE or IO$_SETCHAR. The possible function modifiers are IO$M_READATTN and IO$M_WRTATTN. IO$M_WRTATTN requests notification of an unsolicited message written to that mailbox. An unsolicited message is one written to a mailbox that has no outstanding read request. IO$M_READATTN requests notification when any process requests a read from that mailbox and there is no message in it.

Attention ASTs of each type can be declared by multiple processes for the same mailbox. When a condition corresponding to an attention AST occurs, all ASTs of the appropriate type are delivered. Only the first process to make a corresponding I/O request will be able to complete the transfer of data signaled by the attention ASTs.

Read and write attention ASTs must be reenabled after delivery because the entire attention AST list is delivered and removed after each occurrence of the specified condition.

Chapter *Process Dynamics* describes how DCL uses mailbox attention ASTs to communicate among spawned subprocesses and the top-level process.

8.10.5 Out-of-Band ASTs

The terminal driver uses a newer form of AST mechanism to notify a process that an out-of-band character has been received from its terminal. Out-of-band characters are control characters, the ASCII codes 00 to 20_{16}. (Although CTRL/C and CTRL/Y are in this range, the terminal driver provides the attention AST mechanism described previously to notify a process of their receipt for compatibility with earlier versions of VAX/VMS.) Out-of-band ASTs are similar to attention ASTs in that the terminal driver forks down to IPL$_QUEUEAST to queue an ACB to the process.

The most significant difference between the attention AST mechanism and the out-of-band AST mechanism is that, once declared, out-of-band ASTs are delivered to the process for its lifetime or until the $CANCEL system service is requested to flush the AST list. Another difference is that the out-of-band AST mechanism employs a piggyback special kernel mode AST routine.

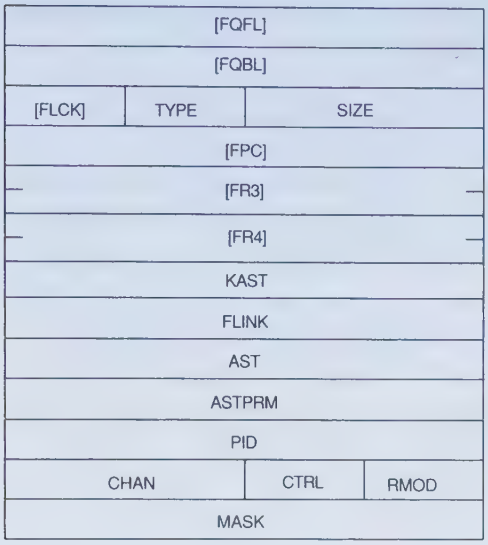
8.10.5.1 The Terminal AST Block

The terminal driver builds a data structure called a terminal AST block (TAST) to describe an out-of-band AST request. Figure 8–5 illustrates the TAST.

The TAST can be in two lists at once because of its structure. Through TAST$L_FLINK, the TAST is always queued to the terminal UCB in a singly linked list. Through the first two longwords of the TAST, it can be inserted into a fork queue or a process's ACB queue. The terminal driver sets the bit TAST$V_BUSY in TAST$B_CTRL when the TAST is in use as a fork block or ACB. The TAST

includes space for fork process context (that is, procedure value of a fork process routine, fork R3, and fork R4) and the AST information (procedure value of the AST procedure and its argument, PID, and RMOD fields).

Figure 8–5 Terminal AST Block



8.10.5.2 Set Out-of-Band AST Mechanism

A process requests out-of-band notification by requesting the $QIO system service, specifying IO$_SETMODE (or IO$_SETCHAR) with the function modifier IO$M_OUTBAND.

The terminal driver's FDT action routine invokes COM$SETCTRLAST, in module COMDRVSUB, which performs the following steps:

1. If the user AST procedure address ($QIO P1 parameter) is zero or the character mask ($QIO P2 parameter) is zero, COM$SETCTRLAST interprets the request as a flush out-of-band AST list request (see Section 8.10.5.4).
2. Otherwise, COM$SETCTRLAST allocates a TAST from nonpaged pool.
3. It then acquires the device lock, raising IPL to UCB$B_DIPL, to synchronize access to the TAST list.
4. COM$SETCTRLAST next scans the list of out-of-band TASTs, searching for one with the same characteristics as the $QIO request. The following items are checked:
   - — The PID. Out-of-band ASTs can be requested for the same terminal device from a process and its subprocesses (which will have different PIDs).
   - — The channel number

5. If COM$SETCTRLAST finds a TAST with the same characteristics that is not in use, it modifies the existing TAST by replacing the AST address and the control mask. It then invokes COM$DRVDEALMEM, in module COMDRVSUB, to create an IPL 6 fork process to deallocate the just-allocated TAST. This unusual sequence is required because COM$SETCTRLAST must hold the device lock while scanning the TAST list. During that time, it cannot allocate pool, synchronization to which is controlled at a lower IPL.

   If the TAST is in use (perhaps queued as an ACB to the process), COM$SETCTRLAST marks it as "lost" and removes it from the list. COM$SETCTRLAST charges the process AST quota and initializes the just-allocated TAST to describe the request. It copies information from the IRP (the AST procedure address, channel number, and PID) and the $QIO character mask into the TAST. It inserts the TAST into the queue position of the lost TAST.

6. If it does not find a similar TAST, it initializes the just-allocated TAST and charges the process AST quota. It places the TAST at the tail of the list.

7. COM$SETCTRLAST ORs the $QIO character mask into the terminal's out-of-band AST summary mask, the field UCB$L_TL_OUTBAND. This mask represents all the control characters for which the terminal driver must deliver an out-of-band AST. It then releases the device lock, restoring the previous IPL.

8.10.5.3 Delivery of Out-of-Band ASTs

When a control key is typed at a terminal, the terminal driver checks whether that control character is represented in the terminal's out-of-band AST summary mask. If the bit in the summary mask is set, an out-of-band AST has been requested for that control character. The terminal driver interrupt service routine invokes COM$DELCTRLAST, in module COMDRVSUB, to deliver the out-of-band AST. The terminal driver uses an alternative entry point, COM$DELCTRLASTP, to specify that only ASTs requested by a particular process be delivered.

The following steps summarize the delivery of out-of-band ASTs:

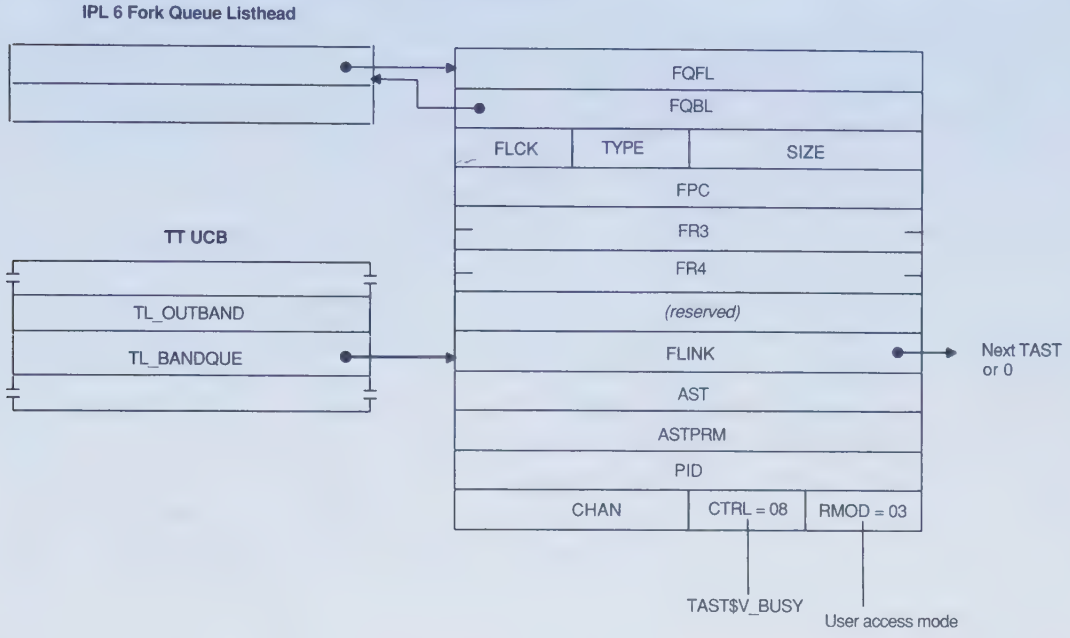
1. COM$DELCTRLAST is entered at device IPL with the device lock held to synchronize access to the TAST list. It scans the list of TASTs for one whose character mask contains the character typed at the terminal.

   When it finds one with a matching character mask, it checks the busy bit to see whether the control block is already in use. In the case of entry through COM$DELCTRLASTP, the routine also compares the PID in the TAST to the requested PID. If they are not equal, the routine goes on to the next TAST in the queue.

   If TAST$V_BUSY is set, COM$DELCTRLAST skips that TAST. If TAST$V_BUSY is clear, COM$DELCTRLAST sets it, marking the TAST in use, and records in TAST$L_ASTPRM the control character that was received.

2. The synchronization considerations previously described for COM$DELATTNAST apply to COM$DELCTRLAST as well. It creates an IPL 6 fork process, using the TAST as an FKB, to queue each AST. The TAST also remains linked to the terminal UCB list of TASTs. Figure 8–6 shows the TAST in the terminal UCB's TAST list and in the fork block queue.

Figure 8–6 TAST Used as a Fork Block



3. When IPL drops below 6, the fork interrupt is granted. The IPL 6 fork dispatcher removes the TAST from the IPL 6 fork block queue and dispatches to COM$DELCTRLAST's fork process.

4. At IPL 6, COM$DELCTRLAST's fork process routine reformats the fork block into an ACB describing the AST procedure and the access mode of the original out-of-band AST request. The no-delete and piggyback special kernel mode AST flags are set in the ACB, and the special kernel mode AST field is loaded with the procedure value of COM$DELCTRLAST's piggyback special kernel mode AST.

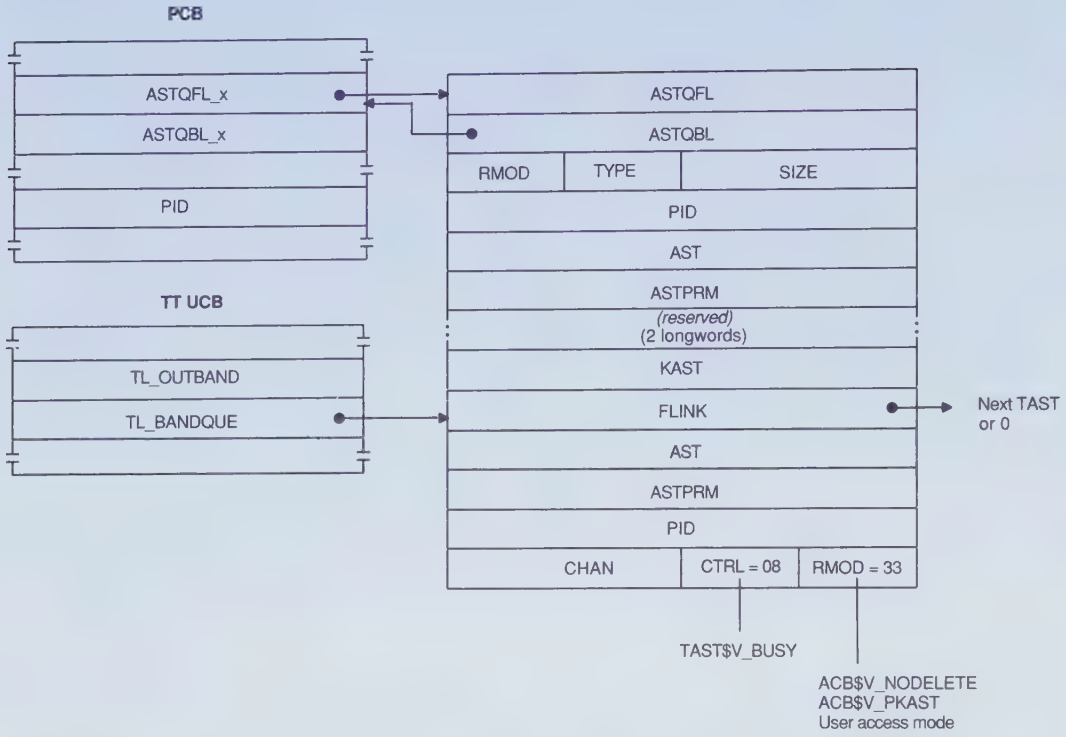
5. The fork process routine invokes SCH$QAST, which acquires the SCHED spinlock and then queues the ACB to the process that requested the attention AST. Figure 8–7 shows the TAST in use as an ACB.

6. When the process receives the AST, the piggyback special kernel mode AST routine is executed first. The piggyback special kernel mode AST performs two functions:

   a. It clears TAST$V_BUSY.

   b. If the TAST is marked as "lost," the piggyback special kernel mode AST routine deallocates it and returns AST quota to the process. A TAST is "lost" when COM$FLUSHCTRLS is unable to deallocate it because its busy bit is set (see Section 8.10.5.4). Once the AST has been delivered, the TAST is no longer needed.

Figure 8–7 TAST Used as an ACB



8.10.5.4 Flushing an Out-of-Band AST List

The list of out-of-band ASTs is flushed as the result of an explicit user request, a $CANCEL, or a $DASSGN request for the associated device.

A user explicitly requests that the out-of-band AST list be flushed by requesting a $QIO set out-of-band AST with an AST procedure value of zero or a character mask of zero (see Section 8.10.5.2). When COM$SETCTRLAST receives such a request, it branches to COM$FLUSHCTRLS.

COM$FLUSHCTRLS is entered with the PID and channel number of the attention ASTs to be deleted. COM$FLUSHCTRLS performs the following operations:

1. It acquires the device lock, raising IPL to UCB$B_DIPL of the device.
2. It scans the out-of-band AST list and compares the PID and channel number in the TAST with those of the requested flush operation. As it scans the list, it builds a new out-of-band AST summary mask. If COM$FLUSHCTRLS finds a TAST that does not match, COM$FLUSHCTRLS ORs its control characters into the summary mask being built and goes on to the next TAST.
3. If the PIDs and channel numbers match, COM$FLUSHCTRLS removes the AST from the list. It checks TAST$V_BUSY to see whether the TAST is in use as a FKB or ACB.

   If TAST$V_BUSY is set, COM$FLUSHCTRLS sets the "lost" bit so that the TAST will be deallocated once its AST is delivered.

If the TAST is not busy, COM$FLUSHCTRLS returns the process AST quota and deallocates the TAST to nonpaged pool.

4. COM$FLUSHCTRLS continues processing until it has scanned the entire list. It then replaces the old summary mask with the one just built.
5. COM$FLUSHCTRLS releases the device lock, restoring the IPL at which it was entered.

8.11 Revision Information

Source Module	Build Date
[SYS]COMDRVSUB.MAR	18-June-1991
[LIB]TASTDEF.SDL	18-June-1991
[LIB]ACBDEF.SDL	18-June-1991
[LIB]PCBDEF.SDL	16-Sep-1991
[LIB]ASTSTKDEF.SDL	16-Aug-1991
[SYS]SYSASTCON.MAR	26-June-1991
[SYS]ASTDEL.MAR	16-Sep-1991
[SYS]ASTDEL_STACK.M64	16-Sep-1991
[SYS]SCHEDULER.M64	16-Aug-1991

Part III/Synchronization

Chapter 9

Synchronization Techniques

"Time," said George, "why I can give you a definition of time.
It's what keeps everything from happening at once."

Ray Cummings, *The Man Who Mastered Time*

In an operating system that allows interrupts, the interrupting code must coordinate, or synchronize, with the code being interrupted to ensure correct behavior. Routines sharing data structures must synchronize modifications to shared data; a thread of execution modifying shared data must synchronize with both read and write accesses by other threads.

A thread of execution sharing data with an I/O processor must synchronize its accesses to the data with those of the I/O processor.

[*Censored*] VMS uses a combination of the following [*censored*] mechanisms and software techniques to synchronize the actions of code threads that might otherwise interfere with each other:

- Atomic memory accesses to aligned longwords and quadwords
- Load-locked (`LDx_L`) and store-conditional (`STx_C`) instructions
- Interrupt priority level (IPL)
- Memory barriers to enforce order on reads and writes of data accessed by multiple processors (CPUs and I/O processors)
- Queue support, provided by privileged architecture library (PAL) routines
- Spinlocks to synchronize access to executive data
- Mutual exclusion semaphores (mutexes)
- Lock management system services
- Event flags
- Parallel processing Run-Time Library routines

Because many of the executive synchronization techniques used by [*censored*] VMS were originally based upon the VAX architecture, this chapter occasionally discusses a VAX synchronization mechanism as background for explaining [*censored*] mechanisms and contrasts VAX and [*censored*] mechanisms.

This chapter deals primarily with synchronizing access to memory. Although access to other types of storage must be synchronized, for example, to shared hardware resources or data in files, the techniques for such synchronization are based upon synchronizing access to memory.

9.1 Overview

Synchronization is a term commonly used to refer to the simultaneous occurrence of two or more events. In a computer context, however, the word is used to refer to the coordination of events. The coordination can still be as specific as the simultaneous occurrence of events; this use of the term occurs most often in descriptions of hardware mechanisms. In descriptions of software, *synchronization* usually refers to the coordination of events in such a way that only one event happens at a time. This specialized kind of synchronization is known as serialization. Serialized events are assigned an order and processed one at a time in that order. While a serialized event is being processed, no other event in the series is allowed to disrupt it.

Atomicity and mutual exclusion are frequently described as different types of serialization, although the two concepts overlap. *Atomicity* refers to the indivisibility of a small number of actions, such as those occurring during the execution of a single instruction or a small number of instructions. *Mutual exclusion* refers to serializing the execution of groups of instructions so that one group completes before another starts.

Algorithms requiring synchronization take many forms and arise in many contexts. Most of them reduce to solving a small number of fundamental problems.

One such problem is the requirement that a thread of execution read or write multiple data items as an atomic operation. If a thread has written some but not all of the items when another thread interrupts and reads the data, the interrupting thread obtains an inconsistent view of the data item. If, on the other hand, a thread has read some but not all of the items when another thread interrupts and writes them all, the interrupted thread gets the inconsistent view.

Another closely related synchronization problem is the requirement that a thread of execution read a data item and write an updated value into the same location. If another thread with the same intent toward that location can intervene after the read and before the write, so that both threads read the same value and both write the location, the change to that location is not atomic: the change made by one of the threads overlays the change made by the other.

The selection of a synchronization method for threads of execution that access a shared data item depends upon issues like the following:

- Which threads of execution can interrupt a thread accessing the data, and which threads can it interrupt?
  - — Do all the threads execute within the same process context?
  - — In which access mode do the threads execute?
  - — Do all the threads execute in process context?
  - — At which IPL do the threads execute?

  - Do all the threads run on the same processor?
- What are the characteristics of the shared data item?
  - In which address space is it?
  - What is its size and alignment?
  - What is the protection of the pages that it occupies?

The sections that follow discuss synchronization issues in more detail: the types of threads of execution, the kinds of memory accesses, *censored* synchronization mechanisms, the software synchronization techniques based on those mechanisms, and the software techniques appropriate to different types of threads of execution.

9.1.1 Threads of Execution

Code threads that can execute within one process include the following:

- Mainline code in an image being executed
- Asynchronous system traps (ASTs) that interrupt the image
- Condition handlers established by the image and entered after exceptions occur
- Inner access mode threads of execution entered as a result of system service, Record Management Services (RMS) service, and command language interpreter (CLI) callback requests

Process-based threads of execution can potentially share any data in P0 and P1 address space and must synchronize access to any data they share. A thread of execution can incur an exception, as a result of which control passes to a condition handler, or receive an AST, as a result of which control passes to an AST procedure. Moreover, an AST procedure can incur an exception, and a condition handler's execution can be interrupted by AST delivery. If a thread of execution requests a system or RMS service, control passes to an inner access mode thread of execution. Code executing in the inner mode can also incur exceptions, receive ASTs, and request services.

Multiple processes, each with its own threads of execution, can execute concurrently. Although each process has private P0 and P1 address space, processes can share data in a global section mapped into each process's address spaces. Synchronizing access to global section data is required because a thread of execution accessing the data in one process can be rescheduled; afterward, a thread of execution in another process can access the same data. Although processes access the same system address space, the protection on system space pages usually prevents outer mode access. However, process-based code threads running in inner access modes can concurrently access data in system space and must synchronize access to it.

Interrupt service routines access only system space. They must synchronize access to shared system space data among themselves and with process-based threads of execution.

A CPU-based thread of execution and an I/O processor must synchronize access to shared data structures, for example, structures containing descriptions of I/O operations to be performed. (The term *I/O processor* includes I/O adapters and I/O widgets, described in Chapter *I/O Architecture*.)

Multiprocessor execution adds synchronization requirements when the threads that must synchronize can run concurrently on different processors. Because a process executes on only one processor at a time, synchronization among threads of execution within a process is unaffected by whether the process runs on a uniprocessor (one with only one CPU) or a symmetric multiprocessing (SMP) system. However, multiple processes can execute simultaneously on different processors. As a result, processes sharing data in a global section may require additional synchronization for SMP system execution. Furthermore, process-based inner mode and interrupt-based threads can execute simultaneously on different processors and may require synchronization of access to system space beyond what would suffice on a uniprocessor.

9.1.2 Reading and Writing Memory

The term *alignment* refers to the placement of a data item in memory. For a data item to be naturally aligned, its lowest addressed byte must reside at an address that is a multiple of the size of the data item in bytes. For example, a naturally aligned longword has an address that is a multiple of 4. An unaligned data item is one whose address is not naturally aligned. Usually, the term *naturally aligned* is shortened to *aligned.*

The term *atomicity* means the indivisibility of one or more actions. When there is more than one action, one of the actions cannot occur by itself; if one occurs, they all occur. The term must be qualified by the viewpoint from which the actions appear indivisible; an operation that is atomic for threads running on the same processor can appear as multiple actions to a thread of execution running on a different processor.

An atomic memory reference is one that results in one indivisible read or write of a data item in memory; no other access to any part of that data can occur during the course of the atomic reference. Atomic memory references are important for synchronizing access to a data item shared by multiple writers or by one writer and multiple readers. References need not be atomic to a data item that is not shared or to one that is shared but only read.

Given a thread of execution trying to write atomically to memory, the basic synchronization issues are as follows:

1. Based on the size and alignment of the data item, can the write be performed as a single memory operation indivisible from any viewpoint? If it can, there is no synchronization issue. If the data item is unaligned, larger than the maximum size of memory access, or smaller than the minimum size of memory access, the write must be performed as multiple memory operations.
2. If multiple memory operations are required to write the data item, are there any other threads of execution that share it? If the data item is not shared, there is no synchronization issue. If the data item is shared, an intermediate state of memory might be visible, or multiple threads might each write part of the data item, resulting in inconsistent data.

3. If other threads of execution share the memory, can each such thread successfully prevent all other threads from accessing the data item during its own access?

   In other words, if the other threads execute on the same CPU, can interrupts or exceptions intervene so that one or more of these threads execute while the data item is in an intermediate state? Can the interrupts or exceptions be blocked, or can the other threads somehow be prevented from accessing the data item in an intermediate state?

4. If the threads accessing the data item execute on multiple CPUs, can each thread prevent the others from accessing the data item in an intermediate state? Also, how can a thread ensure that the others see its writes in order? That is, if they see the last write, does that imply that they see the earlier ones as well? If they write to the same memory locations, can each block the others until its writes complete?

A VAX CPU is required to read or write the following memory operands in a single memory operation that is indivisible from the viewpoint of any other thread of execution:

- Byte operand
- Aligned word operand
- Aligned longword operand
- Bit field contained in one byte
- Aligned longword address used in a displacement deferred mode or autoincrement deferred mode operand specifier. For example, in the instruction `BICL R0,@88(R7)`, if the longword at 88(R7) is aligned, its fetch is atomic.

A VAX CPU is not required to implement access to any other operand types as a single operation. For example, to write an unaligned longword that crosses a quadword boundary, most VAX CPUs would perform two separate memory operations. Nonetheless, the VAX architecture requires that such an access be atomic with respect to other threads of execution on the same processor, if the instruction causing the access is uninterruptible.

The access is not, however, atomic with respect to processes executing on other processors. Another memory command from a different CPU could be issued and carried out between the two commands issued by the first CPU. As a result, another process could read the longword and receive half old contents and half new. This type of inconsistency is known as data incoherency or word-tearing, where *word* refers not necessarily to a 16-bit quantity but to any unit in which the memory is accessed.

Figure 9–1 contrasts writing to aligned and unaligned longwords on a VAX CPU. In Figures 9–1 through 9–3, each hexadecimal digit represents a nibble, that is, half a byte. The string ?? represents the initial indeterminate value of a byte.

Figure 9–1 Writing Aligned and Unaligned Data on a VAX CPU

Part 1 shows a simple aligned longword write by thread A running on a VAX CPU. Part 2 shows the two successive memory operations required for a write to an unaligned longword that crosses a quadword boundary. Part 3 shows how thread B, running on a different processor, can read the longword after it has been only partially updated by thread A. Because the unaligned longword cannot be written in a single memory operation, thread B can read it while it is in an intermediate, inconsistent state, half written by thread A and half still the initial contents.

In contrast to the variety of memory accesses allowed by the VAX architecture, the [*censored*] architecture allows access only to an aligned longword or an aligned quadword. Reading or writing an aligned longword or quadword of memory is atomic with respect to any other thread of execution on the same or other processors. Part 1 of Figure 9–2 shows a simple aligned longword read from memory.

Reading or writing unaligned longwords and unaligned quadwords is possible only through a sequence of instructions. Such a sequence is not atomic from any viewpoint. An instruction sequence that stores an unaligned longword, for example, requires the following steps:

1. Read each quadword in which part of the longword resides.
2. Insert the longword's bytes into each quadword.
3. Write each quadword back to memory.

Example 9–1 shows a sequence of [*censored*] instructions that stores 32 bits of data into an unaligned longword occupying two quadwords. Assuming that the longword is unaligned, the sequence uses load-unaligned and store-unaligned instructions. These instructions transform unaligned addresses into aligned ones by clearing the low-order bits and then loading from or storing to the aligned addresses. The numbers in Example 9–1 correspond to numbered steps in Part 2 of Figure 9–2, which illustrates the effects of load-unaligned and store-unaligned instructions.

When a load or store instruction other than a load-unaligned or store-unaligned one references an unaligned longword or quadword, the processor generates an unaligned data trap. There is a unique exception for each possible instruction type that can incur an unaligned trap. [*Censored*] VMS provides an exception service routine for each type. The service routine is passed the register number and virtual address of the unaligned data and executes an instruction sequence analogous to that shown in Example 9–1.

The VAX and [*censored*] architectures differ in granularity of data access. The phrase *granularity of data access* refers to the size of neighboring units of memory that can be written independently and atomically by multiple processors. Regardless of the order in which the two units are written, the results must be identical. The VAX architecture provides byte granularity: individual adjacent or neighboring bytes within the same longword can be written by multiple threads of execution on one or more processors, as can aligned words and longwords.

Example 9–1 Instructions for Storing Data into an Unaligned Longword

```
; At this point:
; R2   bits <31:0> contain the new data to be stored;
;      bits <63:32> are not relevant
; R28  contains the address of the longword to receive the data;
;      the longword can be unaligned and cross a quadword boundary
;
; Calculate the address of the high-order byte of the longword and
; load the aligned quadword that contains it and possibly other bytes
       LDQ_U   R25, 3(R28)  ❶    ;Load quadword containing the
                                 ; high-order bytes of target
       INSLH   R2,R28,R26        ;Position/isolate the high-order
                                 ; bytes of the new data
       MSKLH   R25, R28, R25     ;Zero the corresponding
                                 ; destination bytes
       BIS     R25, R26, R25     ;Replace them with the new data
       STQ_U   R25, 3(R28)  ❷    ;Store the quadword with updated
                                 ; high-order bytes
;
; Calculate the address of the low-order byte of the longword and
; load the aligned quadword that contains it and possibly other bytes
       LDQ_U   R25, (R28)   ❸    ;Load quadword containing the
                                 ; low-order bytes of target
       INSLL   R2, R28, R26      ;Position/isolate the low-order
                                 ; bytes of the new data
       MSKLL   R25, R28, R25     ;Zero the corresponding
                                 ; destination bytes
       BIS     R25, R26, R25     ;Replace them with the new data
       STQ_U   R25,(R28)    ❹    ;Store the quadword with updated
                                 ; low-order bytes
```

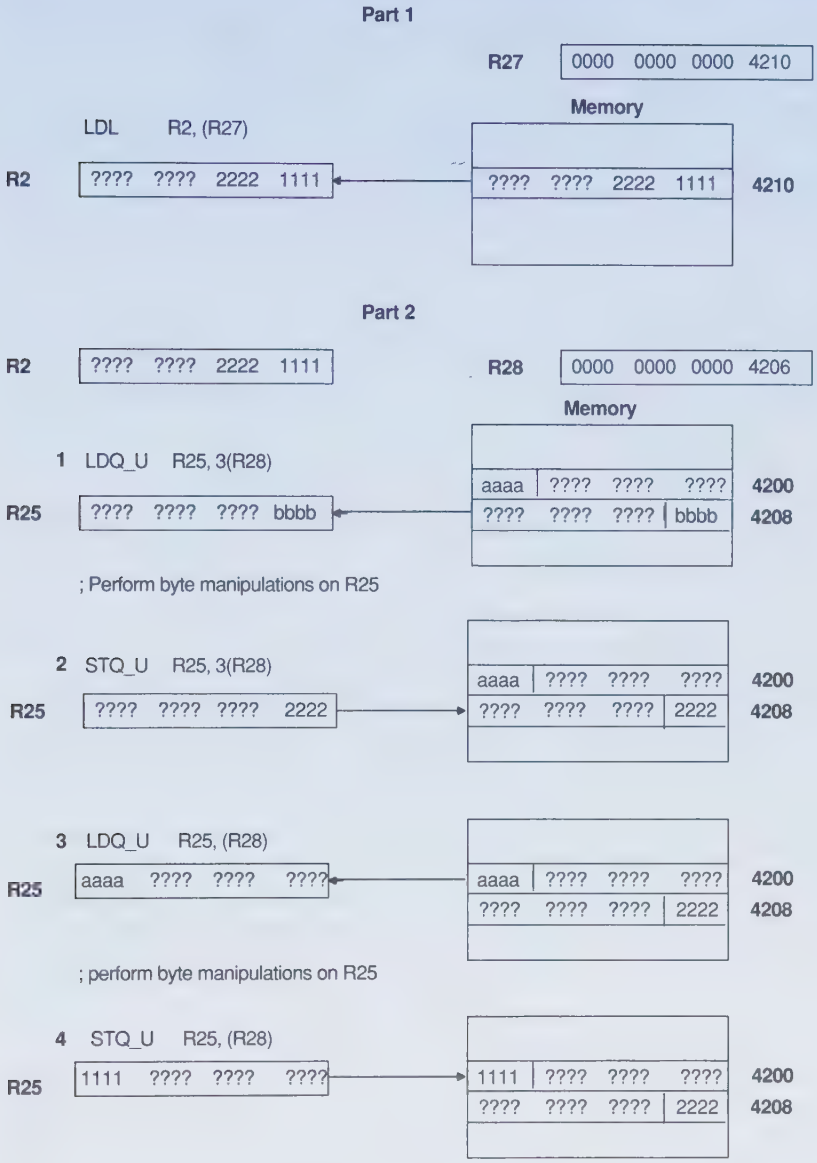
The [*censored*] architecture provides only longword and quadword granularity; that is, only adjacent aligned longwords or quadwords can be written independently. Writing a word or byte requires loading a register with the aligned longword or quadword that it occupies, inserting the word or byte into the register, and writing the register to memory. This multi-instruction sequence is not atomic. Section 9.2.2 shows a method by which the sequence can be made atomic from any viewpoint for data within a single aligned longword or quadword.

The absence of byte and word granularity on [*censored*] CPUs improves CPU performance but has significant implications for access to shared data. In effect, any memory write of a data item other than an aligned longword or quadword must be done as a read-modify-write operation. Furthermore, because the amount of data read and written is an entire longword or quadword, programmers must take particular care to ensure that all accesses to fields within the longword or quadword are synchronized with each other.

Figure 9–3 shows how two threads of execution writing to adjacent words in the same longword can interfere with one another.

Thread A reads the longword containing the word from memory into a register, inserts 2222 into the low-order word in the register, and writes a longword from the register to memory. If thread B concurrently attempted to write 1111 to the other word in the longword, its write or thread A's write would be lost. Whether the end result includes both threads' writes or only one depends on the timing of their loads and stores. Section 9.2.2 shows how the two load and store sequences can be serialized, enabling both writes to have effect.

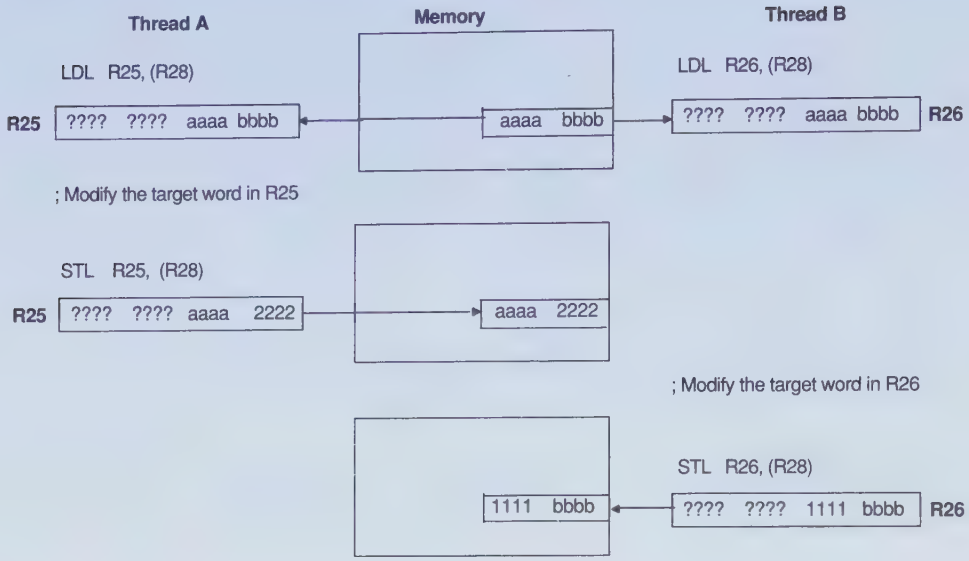
Figure 9–2 Writing Aligned and Unaligned Data on a *Censored* CPU



9.1.3 Read-Modify-Write Memory Operations

A fundamental synchronization primitive for accessing shared data is an atomic read-modify-write operation. This operation consists of reading the contents of a memory location and replacing them with new contents based on the old; any intermediate memory state is not visible to other threads. The VAX and *censored* architectures both provide this synchronization primitive but implement it in significantly different ways.

Figure 9–3 Word Accesses on a *Censored* CPU



The VAX architecture provides a number of uninterruptible instructions, each of which both reads and writes memory within one instruction. When used with an operand type that is accessible in a single memory operation, each provides an atomic read-modify-write sequence. The sequence is atomic with respect to threads of execution on the same VAX processor but not atomic to threads on other processors. For example, when a VAX CPU executes the instruction INCL X, it issues two separate commands to memory: a read, followed by a write of the incremented value. Another thread of execution running concurrently on another processor could issue a command to memory that reads or writes location X between the INCL's read and write. Section 9.2.4 describes read-modify-write sequences that are atomic with respect to threads on all VAX CPUs in an SMP system.

The *censored* architecture provides no single instruction that both reads and writes memory. An atomic read-modify-write operation is only possible through a sequence that includes load-locked and store-conditional instructions, described in Section 9.2.2. Use of these instructions provides a read-modify-write operation on data within one aligned longword or quadword that is atomic with respect to all threads on a single CPU and, if the *censored* system supports SMP in the future, all threads on all CPUs.

9.1.4 Cache and Memory Coherency

Thus far, memory has been treated as a single unit connected to the processor-memory interconnect (PMI). More typically, however, a system has a hierarchy of memory that includes one or more levels of cache. This section provides background information about caches, but not an exhaustive survey of cache protocols or a detailed description of any particular cache protocol.

A cache is a relatively small amount of relatively fast memory that the CPU can access without the delay inherent in sending a command to memory over the PMI. The alternative, requiring the CPU to wait for all memory accesses to complete before proceeding, slows overall system performance dramatically.

Using a cache is an effective strategy for dealing with the speed difference between CPU access and memory access. Some systems have multiple levels of cache. For example, a processor might have a first-level cache on the same chip as the CPU with a minimal access time and a larger, second-level cache on a different chip with an access time longer than that of the first-level cache but considerably shorter than memory access time.

A cache generally includes the contents of memory locations recently accessed by the CPU and those of adjacent locations. Ideally, at any given time, the cache includes the contents of memory locations that the CPU is about to access. The principle is that memory accesses are generally not random but are often related to preceding accesses. For example, an image might access successive elements of an array, or it might execute successive instructions or a loop of instructions.

Before the CPU reads a memory location, it checks whether the target location's contents are cached. If not, it reads that location and adjoining ones from memory and loads them into cache; this is called a cache miss. Whenever the CPU references a location whose contents are in cache (a cache hit), the CPU gets them from the cache, thus accessing them more quickly.

The set of adjoining locations transferred together on a cache miss is called a cache block or cache line. A cache is characterized by the size of its cache blocks, the number of cache blocks it contains, and the mechanism by which the CPU determines whether a particular location's contents are in the cache.

Because cache is considerably smaller than memory, a one-to-one mapping between them is impossible. Instead, a subset of the bits in a memory address identifies a specific cache block (or set of cache blocks, in some cache organizations). Thus, all the memory addresses that have the same value in the subset bits map to the same cache block. Each cache block has an associated valid bit; when the bit is set, the block contains valid data that the CPU can use. Each valid cache block identifies the specific addresses whose contents it includes.

The problem of keeping the contents of cache and memory up-to-date is called cache coherency. On a typical system, the CPU is not the only reader and writer of memory. Even on a uniprocessor system, direct memory access (DMA) I/O operations are performed concurrently with instruction execution. When an I/O processor writes directly to memory locations whose previous contents had been cached, the data that the CPU has cached becomes out-of-date, or stale. On an SMP system, each CPU has its own cache and shares physical memory with the other CPUs. Keeping multiple processors' caches and memory consistent is more complex than uniprocessor cache coherency.

Basic cache coherency issues include the following:

- How is a cache notified that a modification has been made to a location whose contents it holds?
- How does it respond to such notification?

- How does a cache coordinate a write to a location whose contents may be held in other CPUs' caches?

Caches are categorized by the way in which they handle memory writes. On a system with write-through caches, every CPU write to cache goes to memory at the same time. Thus, memory always has a valid copy of the data. In some cache protocols, memory plays an active role in coordinating writes, by sending messages to CPUs to invalidate their cached, stale copies of the data. In others, the writes are broadcast on the PMI, and each cache watches the PMI for invalidates it must perform. An alternative to a simple invalidate message is one that includes the new contents of the modified location so that each cache holding the location can update itself.

On a system with write-back caches, every CPU write goes to its cache. If the data is not in cache, it is first read from memory. The cache is updated, but the new contents are not necessarily written back to memory immediately. Each cache block has an associated "dirty" bit. When the dirty bit is set, it means that the cache block contains modified data not yet written to memory. On a CPU with a write-back cache, when a memory reference results in a cache miss and the address maps to a cache block whose dirty bit is set, the block will be reused but its contents must first be written to memory. Also, because a write-back cache may contain the most up-to-date version of the locations in its cache, it must monitor the PMI for other processors' reads of memory locations it has cached so that it can supply the data.

Write-back caches improve performance by minimizing writes of frequently modified locations. They also reduce PMI traffic, making it possible to connect more processors to the same PMI.

In some protocols, the cache obtains exclusive ownership of the block before updating it by broadcasting a message to any other cache that holds that location; in response, each such cache writes the data to memory and invalidates its cache block. In other protocols, the cache monitors the PMI and records whether any of its blocks' contents are cached by other CPUs. The cache can then treat a shared block as though it were write-through and always write the new contents to memory. Other cache protocols offer additional alternatives.

On an SMP system, processors execute concurrently and read and write memory asynchronously. Caches increase the possibility of delay between the time one processor writes a location and the time the change is propagated and visible to all other processors needing to reference that location. For example, on a system with a write-back cache, there can be a delay between the time a cache inserts a dirty block's contents into a write buffer in preparation for reusing the block and the time the write buffer contents are visible on the PMI as they are written to memory. Furthermore, a PMI write from one processor may not result in an immediate invalidate in a second processor's cache.

A hardware synchronization mechanism that ensures a processor exclusive access to a data item must also ensure that previous changes to it have been processed by the cache and have reached the coherency point.

On a [*censored*] system with multiple processors, there is no common time signal for all processors. Therefore, a term like *most recent* is not meaningful when comparing events on multiple processors. Instead, the *order* in which events (such as reads, writes, interrupt requests) occur is significant. The term *coherency point* refers to the place in a system's memory hierarchy where an order is imposed on reads and writes to a location. The synchronization instructions on a [*censored*] system (`LDx_L`, `STx_C`, and memory barriers) operate at the coherency point. On a [*censored*] system, the coherency point is the PMI.

9.1.5 Order of Reads and Writes

Another strategy for dealing with the difference between memory controller speed and CPU speed is for memory reads and writes to be asynchronous to other steps of instruction execution. On some types of CPU, memory reads and writes are sufficiently asynchronous to not occur in the same order in which they were issued. Although the lack of order does not affect uniprocessor operation, it is significant for any system with more than one processor, even a system with one CPU and one I/O processor. VAX and [*censored*] CPUs differ in the extent to which they perform asynchronous memory reads and writes. More significantly, the architectures differ in whether ordering of reads and writes is required.

VAX CPUs perform reads asynchronously; if the necessary data is not in cache, a command is sent to memory, and processing continues, if possible, on other operands of the instruction. (On a pipelined CPU, operand fetch and evaluation for other instructions may continue.) At the point where instruction execution absolutely requires the read data, instruction processing stalls until the memory operation completes.

Many types of VAX CPU perform memory writes asynchronously, so that while the command and data are being sent to memory, instruction execution can continue. On a VAX CPU that allows only a single asynchronous memory write, a subsequent instruction that attempts another write is stalled until the outstanding write completes. Other types of VAX CPU maximize use of the PMI bandwidth by buffering memory writes, for example, to bytes within an octaword. The assumption behind this approach is that successive writes, for example, to the stack, are often to adjacent memory locations.

Despite the asynchronicity, VAX multiprocessors are required to implement ordering of reads and ordering of writes. That is, if one processor in a multiprocessing system executes instructions that issue writes of multiple data items, the writes must be made to memory in the same order in which the CPU issued them. This means, for example, that when CPU A writes location X and then writes location Y, if CPU B reads Y and gets its new value, it is guaranteed to be able to read the new value in X.

The VAX architecture requires only that reads be ordered with respect to other reads, and writes be ordered with respect to other writes. Some processors make every read and write access to memory in the order in which they were issued, but the architecture does not require it. It permits reads and writes to be disordered with respect to each other.

VAX interlocked instructions, described in Section 9.2.4, order the read and write streams with each other. Without interlocked instructions, certain synchronization algorithms cannot work. A traditional mutual exclusion algorithm, as described by E. Dijkstra, involves two processes running on different processors; each writes its own variable and each reads the other's variable:

Processor A	Processor B
Write X	Write Y
Read Y	Read X

Each processor can read either the old or the new contents of the other's variable. Such an algorithm relies on both processors' having the same view of the lock variables; it can work on a VAX processor only if the accesses are done with interlocked instructions.

Strict ordering of reads and writes impedes the improvement of CPU performance. To improve hardware performance, therefore, the *censored* architecture does not require strict ordering. Instead, a *censored* CPU can reorder memory reads and writes.

Reordering can occur if there are unequal delays for different pieces of data to be written to or read from memory. A pipelined machine can have multiple pipes or unequal pipe lengths. Data traveling through different pipes can have different delay times as a result of different pipe lengths, I/O loading, or memory interference from other processors. A cache can also cause unequal delays if some data stays in the cache longer than other data.

The reordering done by a *censored* CPU is transparent to threads running on that CPU; processes that execute on a single processor can rely on writes from that processor becoming visible in the order in which they are issued. However, multiprocessing applications cannot rely on the order in which writes to memory become visible throughout the system.

For example, writes issued by a CPU can become visible to an I/O processor in an order different from that in which they were issued. Consider an instruction sequence that contains a write to location A followed by a write to location B. Location A is not in cache, but location B is. Because location A is not in cache, it must be read into cache before being written. In the meantime, the write to location B completes. This is called a write hit under a miss.

Another way in which writes can be disordered is when an error occurs. Consider an instruction sequence that contains a write to location A followed by a write to location B. The data written to A is received at the memory controller with an error, but the write of B completes. The CPU resends the write to A, but in the meantime an I/O processor reads A and B, getting an old value of A and a new value of B. To guarantee strictly ordered memory writes, a CPU must delay writing buffers to memory until a previously written buffer has been accepted by memory without parity error. Because such delays decrease performance, *censored* CPUs are not required to delay a subsequent write until the first has been received correctly.

Reads issued by a CPU can complete out of order. Consider an instruction sequence that contains a read from location A followed by a read from location B. Location A is not in the cache, but location B is. This is called a read hit under a miss. After the read of B completes, but before A is read from memory, an I/O processor writes both A and B. Although the CPU's cache is invalidated, it has already loaded the old value of B into a register. As a result, after the read of A completes, the CPU has an old value of B and a new value of A.

A multiprocessing application that assumes read-write ordering and that works on a uniprocessor can fail if it runs on an SMP system. The following example shows two processes, A and B, that share two pieces of data, X and FLAG. Process A calculates a new value for X and then sets FLAG as a signal to process B that it can read the new value of X.

Process A	Process B
Clear FLAG	Read FLAG and test for nonzero
Calculate new value of X	Read FLAG and test for nonzero
Store new value of X	. . .
Store 1 in FLAG	
	If nonzero, read X

If A's writes are visible in the wrong order to process B, then B can read a nonzero FLAG and an old value of X, contrary to the intent of the algorithm.

Care must be taken with application or executive code that shares data with an I/O processor or with another process or thread of execution that could run concurrently on a different CPU in a possible future [*censored*] SMP system. Where strict ordering is required for correct execution, the code must explicitly specify the necessary ordering. Section 9.2.3 describes memory barriers, which provide the basic ordering mechanism.

9.1.6 Instruction Execution

VAX instructions are quite varied—in length, number of operands, number of memory references, operand size, and addressing modes. Some instructions read from memory, some write to memory, and some do both.

The VAX architecture is based on a model of instruction stream processing in which one instruction executes at a time. Basically, an instruction consists of an opcode and all its operand specifiers. Additionally, for certain types of operands, an instruction can include short literals, immediate mode operands, absolute addresses used in absolute mode addressing, branch displacements, and so on.

In processing an instruction, a VAX CPU takes the following steps:

1. Translate the virtual instruction address.
2. Fetch the instruction from memory.
3. Decode the instruction.
4. Fetch the required operands.
5. Execute the instruction.

6. Store results, if any, in the specified locations.
7. Update the program counter (PC).

The CPU completes the execution of each instruction, including all memory accesses, before starting the execution of any part of the next instruction.

Because a strict interpretation of this model yields poor performance, many VAX CPUs overlap the steps of instruction processing. For example, most have instruction lookahead logic that prefetches and decodes one or more instructions from memory during the execution of other instructions. As a result, whenever a thread of execution alters the instruction stream, it must synchronize its alterations with the CPU's execution. Section 9.2.5 describes how this is done.

Whether an instruction's memory references are atomic depends on whether the operands are types to which atomic references are possible, whether the CPU permits interrupts during the instruction's execution, and whether more than one thread can execute concurrently. When only one thread can execute at a time (as in a system with only one CPU and no intelligent I/O processors or controllers), memory references are atomic with respect to other threads of execution on that CPU if interrupts are prevented. For example, the absolute queue instructions INSQUE and REMQUE each make several memory references in manipulating a queue. A VAX CPU allows no interrupts during the execution of these instructions. Thus, the insertion or removal of an element at the head or tail of an absolute queue is atomic with respect to threads of execution on the same processor.

The VAX architecture allows interrupts in only one category of instructions, called first-part-done (FPD) instructions. FPD instructions can be interrupted at well-defined points during their execution; sufficient status is saved in general registers to permit instruction restart at the point of interruption. The VAX architecture specifies that all other instructions are to be uninterruptible in the following sense. If an instruction is interrupted, the microcode must save the software-visible state of the CPU, so that when the interrupt is dismissed, the instruction can restart and execute correctly. This guarantee of restartability for non-FPD instructions generally means that their execution is effectively uninterrupted.

An exception can result from the attempted execution of most types of instruction. An exception is synchronous with attempted instruction execution and is incurred immediately; that is, VAX exceptions are precise. If the exception is one from which recovery is possible, for example, a page fault, the architecture specifies that the software-visible state of the CPU be restored so that the instruction can be restarted from the beginning after the exception is dismissed.

Censored instructions are simpler than VAX instructions. Each *censored* instruction has a fixed form and a fixed length, and can make at most one memory reference, either a load into a register or a store from one. As previously described, no single *censored* instruction performs a read-modify-write operation, and memory references can only be made to aligned longwords and quadwords. (PALcode routines implement several key functions, such as queue support and process context switch, that require multiple memory references. A PALcode routine is invoked with one instruction, runs with interrupts disabled, and may

not incur exceptions; its memory references thus appear atomic with respect to any threads of execution on the same processor.)

In processing an instruction, a [*censored*] CPU takes the following steps:

1. Translate the virtual instruction address.
2. Fetch the instruction from memory.
3. Decode the instruction.
4. Execute the instruction, loading or storing a memory operand, if present.
5. Update the PC.

Unlike the VAX architecture, the [*censored*] architecture permits multiple nonconflicting instructions to be executed concurrently, with results forwarded from one instruction to another. Compilers schedule code sequences that maximize concurrency, with memory accesses separated from one another where possible.

[*Censored*] CPUs have instruction lookahead. As on a VAX CPU, alterations to the instruction stream must be synchronized with the CPU's lookahead and concurrent execution of multiple instructions. Section 9.2.5 describes how this is done.

[*Censored*] instructions are not interruptible.

Exceptions, other than arithmetic ones, are synchronous with attempted instruction execution and are incurred immediately. However, to increase performance, the [*censored*] architecture permits imprecise arithmetic traps. When an arithmetic trap is detected, the PC may have been updated an indeterminate number of instructions past the one whose execution triggered the exception. In other words, an arithmetic exception does not uniquely identify the PC of the instruction that incurred the exception. A thread of execution can force the generation of a pending exception by executing a `DRAINT` or `CALL_PAL DRAINA` instruction. All other exceptions are precise. Chapter *Condition Handling* discusses the imprecision of arithmetic exceptions in more detail.

As previously described, the order in which memory writes from one processor are visible to another, such as an I/O processor, is not necessarily the order in which they were originally written. Software must provide explicit ordering, when necessary, by executing a memory barrier instruction, described in Section 9.2.3.

9.2 Synchronization at the Hardware Level

To assist with synchronization, the VAX architecture provides atomic memory references, uninterruptible instructions, IPL, and interlocked memory accesses. Many read-modify-write instructions, including queue manipulation instructions, are uninterruptible and thus provide an atomic update capability on a uniprocessor. A kernel mode code thread can block interrupt and process-based threads of execution by raising IPL. It can thereby execute a sequence of instructions atomically with respect to the blocked threads. Threads of execution that run on multiple processors of an SMP system synchronize access to shared data with read-modify-write instructions that interlock memory.

These synchronization mechanisms are of particular interest because the [*censored*] VMS executive has been ported from VAX MACRO and BLISS code that uses them. Some of these mechanisms are present in [*censored*] hardware; others have been implemented in [*censored*] PALcode routines.

A [*censored*] processor provides several mechanisms to assist with synchronization. Although all instructions that access memory are uninterruptible, no single one performs an atomic read-modify-write. A kernel mode thread of execution can raise IPL to block other threads on that processor while it performs a read-modify-write sequence or executes any other group of instructions. Code running in any access mode can execute a sequence of instructions containing load-locked (`LDx_L`) and store-conditional (`STx_C`) instructions to perform a read-modify-write sequence that appears atomic to another thread of execution. Memory barrier instructions order a CPU's memory reads and writes from the viewpoint of I/O processors. Additional synchronization mechanisms are provided by PALcode routines that emulate VAX queue and interlocked queue instructions.

The sections that follow describe these mechanisms in more detail.

9.2.1 Interrupt Priority Level

On a uniprocessor system, VMS synchronizes access to systemwide data structures by requiring that all threads sharing data run at the IPL of the highest priority interrupt that causes any of them to execute. Thus a thread's accessing of data cannot be interrupted by any other thread that would access the same data. Section 9.4 describes VMS's use of elevated IPL.

IPL is a processor-specific mechanism; raising IPL on one processor has no effect on another. On an SMP system, code threads running concurrently on different CPUs must synchronize access to shared system data by a method other than IPL.

On a VAX system, code threads running concurrently on different processors synchronize through instructions that interlock memory. Memory interlocks also synchronize access to data shared by an I/O processor and a code thread.

On a [*censored*] system, access to a data structure shared by an I/O processor and a code thread must be synchronized through a load-locked/store-conditional sequence.

9.2.2 LD*x*_L and ST*x*_C [*Censored*] Instructions

Together the `LDx_L` and `STx_C` [*censored*] instructions enable the construction of a code sequence that performs an atomic read-modify-write to an aligned longword or quadword. Rather than blocking other threads' modifications of the target memory, the code sequence determines whether the memory locked by the `LDx_L` instruction could have been written by another thread during the sequence. If so, the sequence is repeated. If not, the store is performed, with the new memory contents moved to the coherency point. If the store succeeds, the sequence is atomic with respect to other threads on the same processor and other processors. These instructions can be executed in any access mode.

To implement these instructions, a processor has two internal registers: one contains a lock flag, and the other, a physical address that identifies the target memory location. These registers describe per-processor state; each member of an SMP system would have its own lock flag and locked physical address register. The amount of memory locked by a LDx_L instruction (called the locked range) is implementation-dependent. The minimum is an aligned quadword. Typically, the amount locked is the size of a cache block, for example, 128 bytes aligned on a 128-byte boundary.

Executing a LDx_L instruction, the processor records the locked physical address and sets the lock flag to indicate that a load-locked/store-conditional sequence is in progress. If an event occurs that could alter a memory location in the locked range after the processor reads it with the LDx_L instruction, the lock flag is cleared. Executing a STx_C instruction, the processor confirms that the lock flag is still set before performing the store; these steps happen atomically to block any event that could alter the state of the flag. The processor copies the lock flag value to the first operand of the STx_C instruction. The code sequence tests the register specified by that operand to determine whether the write occurred.

Example 9–2 shows a typical sequence of *censored* code that contains these instructions.

Example 9–2 Code Sequence Containing Load-Locked/Store-Conditional Instructions

```
; R27 contains the target address
;
RETRY:  LDQ_L   R1,(R27)  ❶       ;Get current contents
        .                         ;Modify R1
        .
        STQ_C   R1,(R27)  ❷       ;Try to store the new value
        BEQ     R1,FAILED ❸       ;Branch if R1 is zero; the store
                                  ; attempt failed
SUCCESS:                          ;The store was successful

        .
FAILED: .                         ;Test for too many failed attempts
        BR      RETRY             ;Repeat attempt
```

The numbers in the example correspond to numbered steps in the following list:

❶ If no exception occurs in the execution of the LDQ_L instruction, the processor loads R1 with the quadword whose address is in R27, records the locked physical address, and sets the lock flag.

❷ Executing the STQ_C instruction, the processor tests whether the lock flag is still set.

— If so, it writes to memory and clears the lock flag. The test and store are atomic.

— If the flag was already clear, the processor does not perform the store.

In either case, it records in the first operand of the STQ_C instruction the state of the lock flag at the time of testing.

❸ The code sequence tests the register identified by that operand to determine what the state of the lock flag was. If the lock flag was set, then the store has succeeded.

If the lock flag was clear, the code sequence must be repeated. The conditional test to initiate the repeat is a branch "forward" in the instruction stream to improve performance for the more likely successful store: on a *censored* processor, backward branches are predicted to be taken, and forward branches are predicted not to be taken.

The code sequence must be coded so as to generate correct results when reexecuted. To avoid infinite loops, the code sequence can keep a count of failed attempts to perform the store and stop trying after a finite number. VMS executive code, however, typically continues to retry on the assumption that the sequence will eventually complete.

The lock flag is cleared by a store to the locked location that originates from outside the processor. To prevent interference by another code thread on the same processor, the architecture also requires that the lock flag be cleared at interrupt or exception initiation and at execution of a `CALL_PAL` instruction. Otherwise, the following sequence could occur:

1. Thread A executes a load-locked instruction to lock location X.
2. Thread B, running as the result of an interrupt, stores to location X; does a load-locked from location X, resetting the locked flag and lock address register; and dismisses the interrupt.
3. Thread A executes a store-conditional to location X and is successful, although it should not be.

The *censored* architecture also permits an implementation to clear the lock flag under the following two circumstances:

- If the processor performs another memory access between the `LDx_L` and `STx_C` instructions
- If the processor branches or otherwise transfers control between executing the two instructions

The code sequence should not include any subsettable instructions, that is, instructions whose implementation on a particular CPU can be through software emulation rather than in hardware. If a particular CPU does not implement some subsettable instructions, executing one causes an exception whose service routine emulates the instruction. This exception would clear the lock flag and prevent the `LDx_L/STx_C` sequence from ever storing successfully.

Furthermore, the code sequence should not be so long that it can incur a timer interrupt at every repeat and thus never be able to complete the store.

Frequently referenced data items, such as spinlocks or semaphores, whose accesses are synchronized through this mechanism should not be located within the same lock range. Concurrent stores by different threads of execution to different data items within the same lock range will lead to failures caused by contention. Optimally, a spinlock or semaphore is located within a lock range containing primarily data that is mostly read and whose access is correlated with ownership of the spinlock or semaphore.

Because the load-locked/store-conditional mechanism can construct read-modify-write sequences that are atomic with respect to threads of execution on all members of an SMP system, it serves several important needs:

- Because the lock flag is cleared by an interrupt or exception, the mechanism can perform an atomic read-modify-write of memory data shared between a mainline thread of execution and an AST procedure or a condition handler.
- The mechanism enables a read-modify-write operation on an aligned longword or quadword that is atomic with respect to writes or read-modify-writes by threads of execution on all processors in the system.
- Furthermore, it enables the sequence of instructions necessary for writing a byte or word of unaligned data (within one quadword) to be atomic with respect to writes from threads of execution on the same or other processors in the system.
- It makes possible [*censored*] emulation of the VAX interlocked instructions, described in Section 9.2.4.

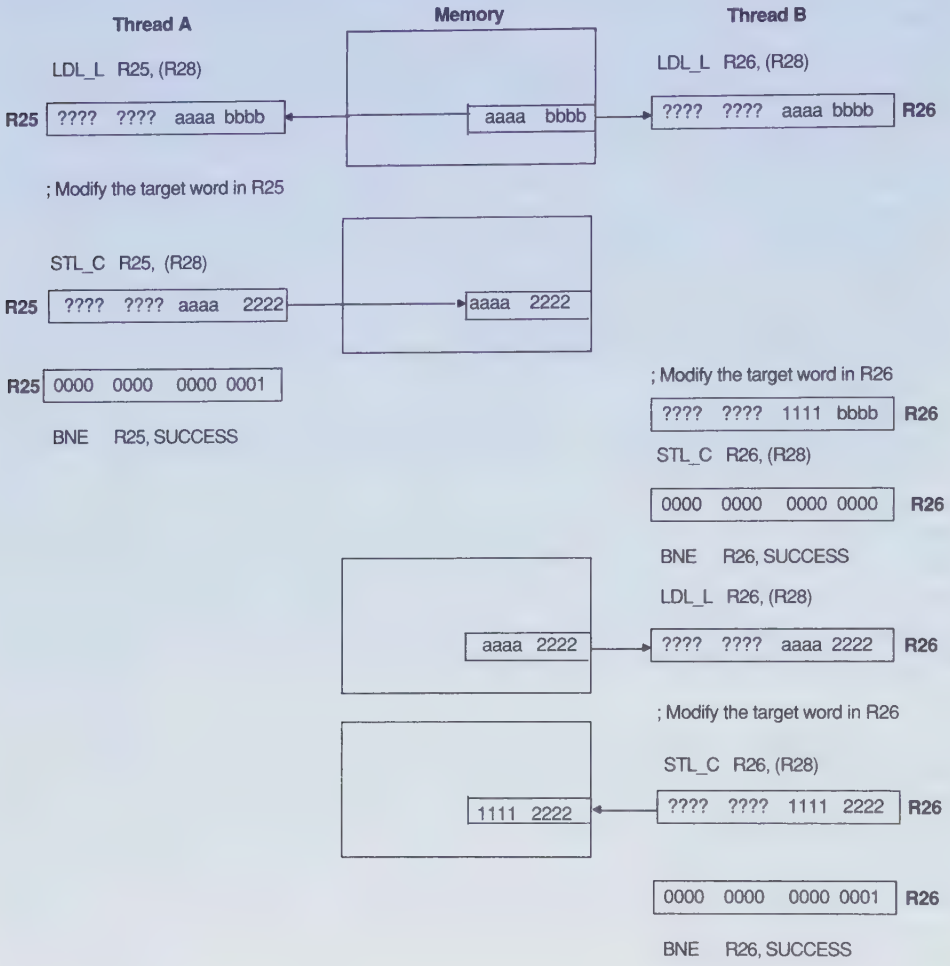
Figure 9–3 showed how two threads of execution trying to write adjacent words in the same quadword can interfere with one another. Figure 9–4 demonstrates how the two writes can be serialized with the use of the load-locked/store-conditional mechanism. In Figures 9–4 and 9–5, each hexadecimal digit represents a nibble, that is, half a byte. The string ?? represents the initial indeterminate value of a byte.

Note that all threads performing read-modify-write operations within the same aligned longword or quadword must use the load-locked/store-conditional mechanism for proper synchronization. Although a thread using this mechanism can detect a concurrent store to the target location, it cannot defend against a second thread that omits load-locked and store-conditional instructions while it does an explicit read-modify-write operation or an implicit one resulting from writing a piece of a longword or quadword. Depending on the timing of the two threads' loads and stores, the memory write of the store-conditional can be overwritten.

Figure 9–5 shows an example of the possible consequences when some threads do not use this mechanism. Thread A executes a load-locked/store-conditional sequence to modify a word within a quadword, the word at location 400, and thread B reads and then writes an unaligned longword, the longword at 402, within the same quadword. If thread B's load happens before A's store and B's store happens after A's store, A's store is lost. With both threads executing on the same processor, once thread A's load-locked occurs, any interrupt or exception that results in entry to another thread will cause thread A's store to fail; as a result, some of the sequences that might otherwise lead to problems will result in A's repeating its load-locked/store-conditional sequence.

To prevent problems caused by desynchronized access to fields within a longword or quadword, Digital recommends aligning data on natural boundaries and "unpacking" shared fields into individual aligned longwords or quadwords. In principle, one shared field that makes up part of a longword or quadword can be

Figure 9–4 *Censored* Load-Locked/Store-Conditional Mechanism



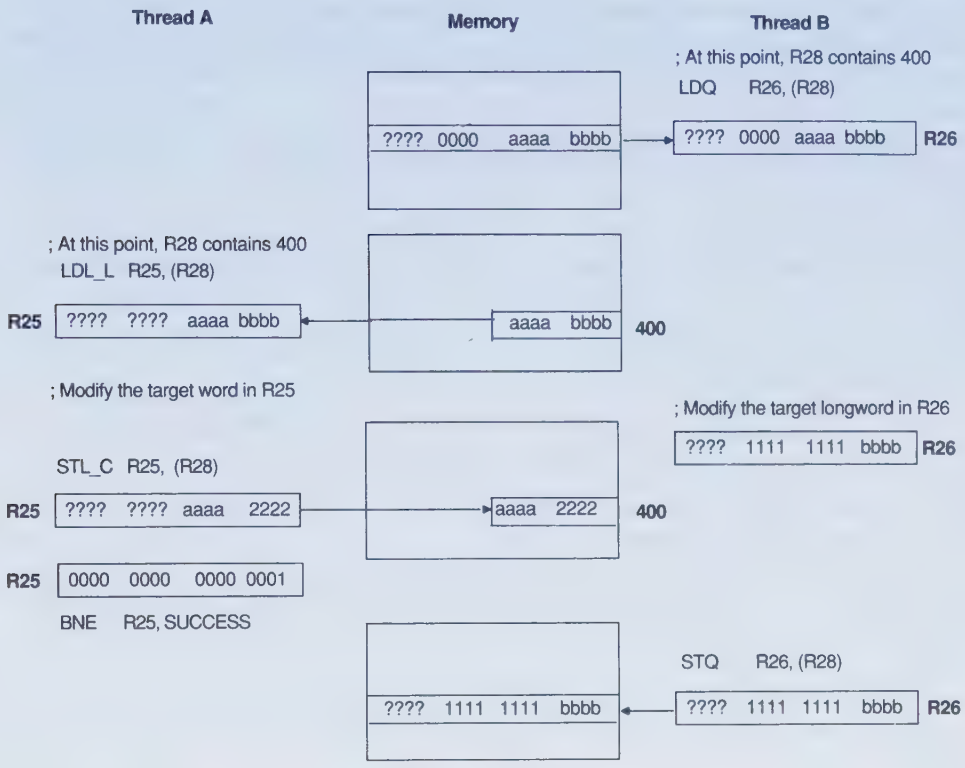
accessed without synchronization problems. However, in practice, a compiler can generate unsynchronized read-modify-write operations on data in other fields in the longword or quadword, affecting the shared field as well. If unpacking shared fields is not possible, an alternative is to add appropriate compiler directives to force atomic access to both the shared and nonshared fields.

9.2.3 Memory Barriers

There are no implied memory barriers in the *censored* architecture except those performed by the PALcode routines that emulate the interlocked queue instructions. Explicit memory barriers must be inserted into code wherever necessary to impose an order on memory references.

On a uniprocessor an instruction's read of a memory location always returns the data from the most recent write access to the same location (unless the memory has been overwritten by DMA I/O). However, as described in Section 9.1.5, reads and writes issued by a processor can complete out of order from the viewpoint

Figure 9–5 Desynchronized Access to Fields in a Quadword



of an I/O processor. Memory barriers are required for ordering reads from and writes to data shared with an I/O processor. If a [*censored*] system supports multiprocessing, memory barriers would also be required for ensuring the order in which other members of an SMP system see writes to shared data and the order in which reads of shared data complete.

There are two types of memory barrier:

- The MB instruction
- The instruction memory barrier (IMB) PALcode routine, which also flushes prefetched instructions that may have been overwritten by a write of the instruction stream (see Section 9.2.5).

Executing a memory barrier forces every preceding write issued by the CPU to reach its coherency point. This does not mean that the memory writes have actually been done when the instruction execution completes; it means that the order of the eventual writes is committed. For example, it may mean that the write commands and data are acknowledged as having been received with good parity at a queue to the memory board; the actual writes may happen later.

Executing a memory barrier also forces all queued cache invalidates to be delivered to the cache before any subsequent read or write is started. Otherwise, a read could be satisfied from cache with stale data, or a write could modify cache, only to be effectively overwritten by a later-delivered invalidate. Executing a memory barrier prevents cache hits under misses; the processor waits for an earlier miss to complete before a later hit, thus ensuring that the invalidates have occurred.

Note that a memory barrier only affects the memory accesses of the CPU that executes it.

The VMS executive and device drivers must access a memory-resident data structure shared with an I/O processor as though it were shared with another CPU. In particular, a memory barrier is necessary in the following cases:

- After updating a data structure and before setting a flag to indicate that the data structure is valid
- After observing a newly updated flag set by an I/O processor and before accessing the data structure whose state the flag represents

Following are several examples of the use of memory barriers to order access to data shared with an I/O processor:

- After updating a memory data structure, a driver executes an MB instruction before writing an I/O processor's interface register to inform it that the structure has been updated.
- A driver separates multiple consecutive accesses to I/O processor interface registers with MB instructions if the register accesses must be seen by the I/O processor in order.
- To access a remote I/O processor's interface registers through an I/O mailbox (see Chapter *I/O Architecture*), a driver initializes fields in a hardware mailbox data structure and executes an MB instruction to ensure that the data structure writes complete before the driver initiates the hardware mailbox operation.
- When a driver and an I/O processor share a command ring buffer with a pointer, the driver must execute an MB instruction after writing a new command in the ring buffer and before updating the pointer.

9.2.4 Interlocked Instructions

The VAX architecture provides seven instructions that interlock memory. The interlock is implemented at the level of the memory controller. A memory interlock enables a VAX CPU or I/O processor to make an atomic read-modify-write to a location in memory shared by multiple processors. The instructions, whose operations are described in detail in the *VAX Architecture Reference Manual*, are the following:

- ADAWI—Add aligned word, interlocked
- BBCCI—Branch on bit clear and clear, interlocked
- BBSSI—Branch on bit set and set, interlocked

- INSQHI—Insert entry into queue at head, interlocked
- INSQTI—Insert entry into queue at tail, interlocked
- REMQHI—Remove entry from queue at head, interlocked
- REMQTI—Remove entry from queue at tail, interlocked

(For simplicity, the description of the interlocked instruction mechanism that follows assumes that the interlock is implemented by the memory controller, as it is in many, but not all, VAX systems and that the cache is write-through so that memory has up-to-date contents.) When a VAX CPU executes an interlocked instruction, it first issues an interlock-read command to the memory controller, bypassing cache. The memory controller sets an internal flag and responds with the requested data. While the flag is set, the memory controller stalls any subsequent interlock-read commands for that same aligned longword from other CPUs and I/O processors, although it continues to process ordinary reads and writes. (Because interlocked instructions are noninterruptible, they are atomic with respect to threads of execution on the same processor.)

When the VAX processor executing the interlocked instruction issues a write-unlock command, the memory controller writes the modified data back and clears its internal flag. The memory interlock persists for the duration of only one instruction. That is, execution of an interlocked instruction includes paired interlock-read and write-unlock memory controller commands.

Synchronizing data with interlocks requires that all accessors of that data use them. In other words, the memory references of an interlocked instruction are atomic only with respect to other interlocked memory references.

The granularity of the interlock is VAX-implementation-dependent. For example, on some processors, while an interlocked access to a location is in progress, no interlocked access to any other location in memory is allowed. The VAX architecture guarantees only aligned longword granularity.

An interlocked queue instruction does its interlocking in two stages. In the first stage, the VAX processor issues an interlock-read command to read the forward link. It then makes a bit test and issues a write-unlock command, setting and checking the low-order bit of the forward link. (Self-relative queue elements are constrained to be quadword-aligned; the low-order three address bits are thus available for other uses.) If the low-order bit was clear, the processor continues with the second stage of the instruction.

During execution of the interlocked queue instruction, the queue itself is interlocked by the low-order bit in the forward link, but only the queue is interlocked; the rest of the memory that would otherwise be interlocked is free. The use of this "secondary interlock" reduces memory interlock contention.

If the low-order bit of the forward link was already set, the processor sets the C condition code bit and completes instruction execution without performing any queue manipulations. The code containing the interlocked queue instruction is expected to test the C bit and, if it is set, try to execute the instruction again. After a number of failures, the queue is presumed corrupt.

The [*censored*] architecture provides no single instruction that both reads and writes memory nor does it include a mechanism to interlock memory against other interlocked accesses. Instead, the load-locked/store-conditional mechanism, described in Section 9.2.2, makes possible a sequence of instructions that performs an atomic read-modify-write. The VAX MACRO compiler generates load-locked/store-conditional sequences for ADAWI, BBSSI, and BBCCI statements. Each sequence begins and ends with an MB instruction.

In code generated for a BBSSI, for example, the first MB ensures that the quadword containing the bit is not read or changed until memory reads and writes issued earlier by this processor have occurred and that any pending cache invalidates have been processed. The second MB ensures that the quadword containing the bit is written back to memory before any subsequent memory reads or writes are issued, for example, of fields in data structures protected by the bit.

On a VAX CPU, an interlocked instruction is the only way to perform an atomic read-modify-write on a shared piece of data. On a [*censored*] CPU, the load-locked/store-conditional mechanism serves that need. For [*censored*] VMS, some interlocked instructions in the executive have been replaced by code sequences that generate only the load-locked/store-conditional sequence without the MB instructions. The MB instructions are not necessary in cases where the atomic read-modify-write is an end in itself rather than a means to implement a semaphore.

A sequence to implement an interlocked queue instruction, however, requires more than a load-locked/store-conditional sequence. Although testing and setting the secondary interlock bit could be done with a load-locked/store-conditional sequence, once a thread of execution has set the bit, its insertion or removal of an element from the queue must complete without interruption to prevent deadlocks. Consider, for example, a thread that succeeds in setting the secondary interlock bit and that is then interrupted by a higher priority thread that loops trying to acquire the secondary interlock itself: the two threads would deadlock. The interlocked queue instructions are therefore emulated by PALcode routines that execute with all interrupts blocked. Furthermore, these PALcode routines begin and end with MB instructions to order reads and writes.

Typically, VAX/VMS and [*censored*] VMS executive code written in VAX MACRO performs interlocked queue manipulations through invocation of one of the following macros: $INSQHI, $INSQTI, $REMQHI, and $REMQTI. The macros include a retry loop in case an access to the queue is already in progress. A sample invocation of $INSQTI and its expanded VAX MACRO code follows:

```
;The VAX MACRO macro invocation
        $INSQTI (R3),G^IOC$GQ_POSTIQ ;Insert packet on queue

;Its expanded code
        CLRL    R0
30000$: INSQTI  (R3),G^IOC$GQ_POSTIQ
        BCC     30001$
        AOBLSS  #900000,R0,30000$
        BUG_CHECK BADQHDR,FATAL
30001$:
```

9.2.5 Synchronizing Access to the Instruction Stream

As previously described in Section 9.1.6, one part of the CPU prefetches and evaluates instructions while another part executes previously fetched instructions. As a result, any write to memory locations containing instructions must be synchronized with the execution of the instructions. Otherwise, it is indeterminate which instructions are executed: old ones, new ones, or a mix. Basically, the prefetched instructions must be discarded, or flushed, and refetched to ensure that the instruction modifications take effect. Both the VAX and [*censored*] architectures provide a way to flush instruction prefetch.

On a VAX processor, a write to the instruction stream must be followed by an REI instruction. For example, when the debugger sets a breakpoint at an instruction by replacing the opcode with a BPT instruction, it must execute an REI instruction to flush instruction prefetch. Similarly, when it removes the breakpoint by replacing it with the original instruction, it must also execute an REI. When XDELTA executes on a processor in an SMP system, the other processors execute an idle loop waiting for XDELTA to complete. When it does, they each must execute an REI instruction to flush possible prefetch of instructions and operands altered as a result of the XDELTA activity.

Another example occurs during the loading of a loadable executive image or a device driver. The nonpaged parts of a loadable executive image are read into pages whose page table entries' valid bits have already been set; a device driver is loaded into valid nonpaged pool pages. After the I/O completes, the loader routine must execute an REI instruction. Moreover, on an SMP system, it must request an interprocessor interrupt so that each processor executes an REI instruction. Since most images are paged in, that is, read into pages whose valid bits are clear, flushing instruction prefetch is not required; the CPU does not prefetch from an invalid page. VAX/VMS executive code written in VAX MACRO invokes the INVALIDATE_VIC macro to perform the REI and to request an interprocessor interrupt for the other SMP members to execute an REI instruction.

On a [*censored*] processor, a write to the instruction stream must be followed by a CALL_PAL IMB. The IMB PALcode routine ensures coherency between prefetched instructions and modifications of the instruction stream. In addition, it provides the same ordering effects as the MB instruction, described in Section 9.2.3.

For example, a debugger running on a [*censored*] processor executes a CALL_PAL IMB instruction after setting a breakpoint by overwriting an instruction with a CALL_PAL BPT. The loader routine that reads nonpageable portions of executive images must also execute a CALL_PAL IMB instruction before it returns to its invoker. [*Censored*] VMS executive code written in VAX MACRO invokes the INSTRUCTION_MB macro to perform the CALL_PAL IMB. If SMP support were enabled, the macro would also request an interprocessor interrupt so that other members would execute a CALL_PAL IMB instruction.

9.3 Synchronization at the Software Level

VMS uses the synchronization primitives provided by the hardware as the basis for several different synchronization techniques. The following sections summarize VMS synchronization techniques, both those available to application software and those used by the executive.

9.3.1 Synchronization Within a Process

Per-process space data is typically process-private and accessible only to threads of execution running within process context. Because only one thread of execution can execute within a process at a time, synchronization of threads executing simultaneously is not an issue. However, delivery of an AST or the occurrence of an exception can intervene in a sequence of instructions in one thread of execution. Therefore, application design must take into account the need for synchronization with condition handlers and AST procedures.

As previously described in Sections 9.1.2 and 9.1.3, accessing unaligned data, writing bytes or words, or performing a read-modify-write operation each requires a sequence of [*censored*] instructions. If the sequence incurs an exception or is interrupted by AST delivery, another process code thread can run. If that thread accesses the same data, it can read incompletely written data or cause data corruption. Aligning data on natural boundaries and unpacking word and byte data reduces this risk.

An application written in a language other than VAX MACRO must identify to the compiler data accessed by any combination of mainline code, AST procedures, and condition handlers to ensure that the compiler generates code atomic with respect to other threads. Similarly, data shared with other processes must be identified.

Furthermore, as described in Section 9.1.2, it is possible for a compiler inadvertently to desynchronize access to a piece of shared data with the code it generates for access to other fields in the same quadword.

A related requirement, not unique to [*censored*] programs, is identifying variables that are modified by routines external to the source module.

Synchronizing access to multiple pieces of data or to one piece of data that occupies multiple quadwords requires use of a higher level technique, such as acquiring ownership of a semaphore during access to the data.

In the case of process-private data accessed from both AST and non-AST threads of execution, the non-AST thread can block AST delivery through the Set AST Enable ($SETAST) system service. Code running in kernel mode can also raise IPL to block AST delivery. The concept of AST reentrancy and other ways of achieving it are described in the *Guide to Creating VMS Modular Procedures*.

Data accessed by an inner mode thread of execution is usually in pages whose protection prohibits access by any outer mode.

9.3.2 Multiprocess Application Synchronization

VMS provides several ways to synchronize multiprocess applications. These include common event flags, lock management system services, and the parallel processing Run-Time Library procedures.

VMS provides basic event synchronization through event flags. Common event flags can be shared among cooperating processes running on a uniprocessor or an SMP system. The processes must be in the same UIC group. A shared, or common, event flag can represent any event detectable and agreed upon by the

cooperating processes. Chapter *Event Flags* describes the implementation of event flags and a technique for using them in multiprocess synchronization.

The lock management system services (Enqueue Lock Request, $ENQ, and Dequeue Lock Request, $DEQ) provide multiprocess synchronization tools that can be requested from all access modes.

The parallel processing Run-Time Library procedures provide support for a number of different synchronization techniques suitable for user access mode applications. These techniques include

- Mutual exclusion implemented through an application-created semaphore or spinlock
- Event synchronization, by which one or more processes can wait for the occurrence of a user-defined event that is triggered by another process
- Barrier synchronization, by which multiple processes wait until a specified number of them have all reached a designated point in their execution

The *VMS RTL Parallel Processing (PPL$) Manual* describes these procedures and their use.

Synchronization of access to shared data by a multiprocess application should be designed to support processes executing concurrently on different members of an SMP system. This strategy results in an application that works on current [*censored*] systems and that will still work if [*censored*] VMS supports such a configuration in the future. Applications that share a global section can use VAX MACRO interlocked instructions or their equivalent in other languages to synchronize accesses to data in the global section. They can also use the lock management system services for synchronization.

9.3.3 Executive Synchronization Techniques

Table 9–1 contrasts the synchronization techniques most commonly used by the VMS executive. These techniques are based upon the synchronization primitives described in earlier sections; in some cases, the executive simply uses the synchronization primitives provided by the architecture.

The techniques listed in the table are intended for synchronizing the accesses of process-based threads executing in multiple processes or of process-based threads and interrupt-based threads. Synchronization based on lock management system services is the only method available to any access mode other than kernel.

VAX/VMS was originally designed to run on a uniprocessor. Traditionally, VAX/VMS synchronized access to system data structures using IPL. However, IPL alone is insufficient to synchronize access to data structures in memory shared by the multiple CPUs of an SMP system. To extend to an SMP environment the uniprocessor synchronization provided by IPL, VAX/VMS Version 5 introduced a mechanism called a spinlock.

In VAX/VMS Version 5 and subsequent versions, a thread of execution not only raises IPL to block interrupts on the same processor but also acquires a spinlock to block concurrent access by other processors. Each systemwide database that could be accessed concurrently is associated with a spinlock.

Because it is derived from VAX/VMS Version 5.4, [*censored*] VMS code also invokes macros that lock and unlock a spinlock to synchronize access to the database protected by the spinlock. However, [*censored*] VMS Version 1 does not support SMP, and its spinlock acquisitions and releases amount to little more than raising and lowering IPL. The hooks for SMP support were left in place to minimize the changes needed between VAX/VMS Version 5.4 and [*censored*] VMS Version 1 and to facilitate development of possible future SMP support. Section 9.5 describes spinlocks in detail.

Table 9–1 Characteristics of Executive Synchronization Methods

Characteristic	IPL	Spinlocks	Mutexes	Locks
Available to outer modes	No	No	No	Yes
Usable from process context	Yes	Yes	Yes	Yes
Usable from system context	Yes	Yes	Yes[1]	Yes[2]
Kinds of sharing	Exclusive	Exclusive	Multiple readers or one writer	Varied modes
Creation	n/a	Most fixed; some dynamic	Most fixed; some dynamic[3]	Dynamic

[1]Mutexes are used almost entirely for process context synchronization. The I/O database mutex is the only one currently locked from system context.

[2]Although a fork-level interface to lock management system services exists, it is currently unused on [*censored*] VMS.

[3]Most mutexes are fixed. Several data structures, however, have a field containing a mutex to synchronize access to other fields in the data structure. These mutexes are created dynamically with the data structures that contain them.

Accesses to shared system data structures by multiple processes from IPLs below 3 can be synchronized by mutexes. Section 9.7 describes mutexes. The VMS executive uses lock management system services to provide systemwide multiprocess synchronization for components like RMS, the file system, the job controller, the device allocation routines, and the Mount utility. Appendix *Lock and Resource Use by VMS Components* describes some of these uses. The lock management system services are described in the *VMS System Services Reference Manual*; Chapter *Lock Management* in this book describes their internal workings.

Another important synchronization issue for VMS involves disk storage. Data structures on a shared disk (for example, files and records within files and the actual disk structure) are protected by lock management system services.

9.4 Elevated IPL

Raising IPL on a processor blocks all interrupts on that processor at the specified IPL value and all lower values of IPL. The traditional VMS method of synchronizing access to system data has been to raise IPL to a high enough level to block all interrupts whose service routines read or write that data. For example, access to the variable-length nonpaged pool list is synchronized at IPL 11, the IPL of the highest interrupt thread from which nonpaged pool allocation is permitted. At IPL 11, all fork process interrupts are blocked, but higher priority software and hardware interrupts can still be granted.

The IPL, stored in the processor status (PS) register bits <12:8>, is altered when the IPL processor register is altered. The register can be altered through the MTPR PALcode routine, which is usually invoked through the VAX MACRO statement MTPR or through a compiler built-in. Typically, VMS executive code alters IPL by invoking the SETIPL or DSBINT macro. Their macro definitions, somewhat simplified, follow:

```
.MACRO  SETIPL  IPL = #31
        MTPR    IPL,S^#PR$_IPL
.ENDM SETIPL

.MACRO  DSBINT  IPL = #31, DST = -(SP)
        MFPR    S^#PR$_IPL,DST
        MTPR    IPL,S^#PR$_IPL
.ENDM   DSBINT
```

The SETIPL macro changes IPL to the specified value. If no argument is present, IPL is elevated to 31. This macro is invoked when the IPL will later be explicitly lowered with another SETIPL or simply as a result of executing an REI instruction. That is, the value of the saved IPL is not important to the routine that is using the SETIPL macro.

The DSBINT macro saves the current IPL before elevating IPL to the specified value. If no argument is present, IPL is elevated to 31. If no alternative destination is specified, the old IPL is saved on the stack. This macro is usually invoked when a later sequence of code must restore the IPL to the saved value with the ENBINT macro. ENBINT, the counterpart to the DSBINT macro, restores the IPL to the value in the designated source argument.

The successful use of IPL as a synchronization tool requires that IPL be raised (not lowered) to the appropriate synchronization level. Lowering IPL defeats any attempt at synchronization. Note, however, that a thread of execution may raise and then lower its IPL as long as it does not lower IPL below that of its entry or below that associated with any spinlocks it holds.

Suppose a thread of execution modifying more than one location in a shared database raises IPL to *x* to block interrupts from other accessors of the database. The first thread of execution is interrupted, after partly making its modifications, by a second thread running in response to a higher priority interrupt. The shared database is now in an inconsistent state. If the second thread were to lower IPL to *x* in a mistaken attempt to synchronize access to the database, it could receive incorrect data or corrupt the database.

Integrity of the database would, however, be maintained if the second thread of execution were to reschedule itself to run as the result of an interrupt at or below *x* and access the database from the rescheduled thread. Forking is the primary way in which an interrupt thread of execution reschedules itself to run at a lower IPL. Chapter *Software Interrupts* describes forking in more detail.

The sections following briefly describe the synchronization use of various IPLs. Note, however, that most of the SETIPL, DSBINT, and ENBINT macro invocations in the executive have been replaced by invocations of macros that acquire and release spinlocks. Each of the IPLs traditionally used for synchronizing access to shared data now has one or more spinlocks associated with it. On a uniprocessor system, the act of acquiring a spinlock is transparently reduced to raising IPL to that of the spinlock. Section 9.5.5 describes the use of each spinlock. From the perspective of a uniprocessor system, those sections can be interpreted as describing the synchronization use of the spinlocks' IPLs.

The macro $IPLDEF defines symbolic names for IPL values.

9.4.1 IPL$_POWER (31)

Kernel mode code raises IPL to IPL$_POWER, or 31, to block all interrupts, including power failure, an IPL 30 interrupt, and machine check, an IPL 31 interrupt. IPL is raised to this level only for a short period of time once the system has been initialized. IPL$_EMB and IPL$_MCHECK are synonyms for IPL$_POWER; they are two different names for the same spinlock.

- A device driver raises IPL to 31 to prevent a powerfail interrupt from occurring, just before it stalls, waiting for an interrupt.
- The entire bootstrap sequence operates at IPL 31 to put the system into a known state before allowing interrupts to occur.
- As described in Section 9.5.5.17, error log buffer allocation and deallocation occur at this IPL.
- As described in Section 9.5.5.16, machine check exceptions and interrupts and parts of the CPU-specific error interrupt service routines execute at IPL 31.
- XDELTA, the executive debugger, runs at IPL 31.

9.4.2 IPL$_HWCLK (22)

When IPL is raised to IPL$_HWCLK, interval timer interrupts are blocked. Section 9.5.5.14 describes the use of the associated spinlock.

9.4.3 Device IPLs (20–23)

A device driver raises IPL to the level at which its associated device interrupts. Raising IPL prevents the device from interrupting while its device registers are being read or written (see Section 9.5.3).

9.4.4 IPL$_PERFMON (15)

IPL$_PERFMON is the IPL associated with the PERFMON spinlock, which synchronizes access to performance monitoring data (see Section 9.5.5.12).

9.4.5 Fork IPLs (6, 8–11)

The executive uses fork IPLs to synchronize access to unit control blocks (UCBs). UCBs are accessed by device drivers and by process-based code, such as the Queue I/O ($QIO) and Cancel I/O on Channel ($CANCEL) system services.

A device driver also uses its associated fork IPL as a synchronization level when accessing data structures that control shared resources, such as multiunit controllers. For this synchronization to work properly, all devices sharing a given resource must use the same fork IPL.

Fork processing, the technique whereby a device driver lowers IPL below device interrupt level in a manner consistent with the interrupt nesting scheme, also uses the serialization technique described in Section 9.6.

9.4.6 IPL 8

IPL 8 is the IPL at which the software timer routine executes. This routine services timer queue entries (TQEs) and handles quantum expiration. The software timer interrupt is requested and granted at IPL 7, but the interrupt service routine raises IPL and runs primarily at 8. Chapter *Time Support* gives further details.

This is the level to which IPL must be raised for any routine to access several systemwide data structures, for example, the scheduler database. By raising IPL to 8, all other interrupt service routines on that processor that might access the same systemwide data structure are blocked from execution until IPL is lowered.

This is also the IPL at which most driver fork processing occurs. While the processor is executing at this IPL, certain systemwide events, such as scheduling and I/O postprocessing, are blocked. However, other more important operations, such as hardware interrupt servicing, can continue.

Traditionally in VAX/VMS, IPL$_SYNCH is the symbolic name for this IPL. In *censored* VMS, synonyms for IPL$_SYNCH with the same numeric value are more commonly used. Each of the following synonyms is associated with its own spinlock: IPL$_MMG, IPL$_SCHED, IPL$_FILSYS, IPL$_TIMER, IPL$_JIB, and IPL$_IO_MISC. IPL$_SCS and IPL$_IOLOCK8, which are also synonyms of IPL$_SYNCH, are two different names for the same spinlock.

In early versions of VAX/VMS, the value of IPL$_SYNCH was 7. In VAX/VMS Version 4, its value was changed to 8 to enable three executive components to run at the same IPL: the distributed lock manager, system communication services (SCS), and the CI port driver.

9.4.7 IPL$_QUEUEAST (6)

When IPL$_SYNCH had a value of 7, device drivers and other high IPL threads of execution that needed to access data, such as the scheduler database, forked to IPL 6 so that they could raise IPL to IPL$_SYNCH.

The terminal driver, for example, might notify a requesting process of unsolicited input or a CTRL/Y through an AST (see Chapter *ASTs*). Queuing an AST to a process requires scheduler database modifications, which must be made at IPL$_SYNCH. The IPL 7 interrupt could not have been used to achieve the same result because it is reserved for software timer interrupts. Thus, this synchronization technique used the first free IPL below 7, the IPL 6 software interrupt. IPL 6 was named IPL$_QUEUEAST, since its primary use as a fork IPL was AST enqueuing.

As a result of changing IPL$_SYNCH to 8, IPL$_QUEUEAST forking is generally unnecessary for serializing access to databases synchronized at IPL$_SYNCH. Fork processes running at IPL 8 can remain at 8; device interrupt service routines and fork processes running at IPLs above 8 can fork to 8. However, many instances of IPL$_QUEUEAST fork processing remain in VMS, unchanged from earlier versions. Executing these operations at IPL$_QUEUEAST rather than at IPL 8 results in placing a somewhat higher priority on IPL 8 fork processing, which is typically I/O processing.

9.4.8 IPL$_RESCHED (3)

IPL$_RESCHED is the IPL of the rescheduling interrupt, whose service routine removes the current process from execution and selects another process to execute. Kernel mode code running in process context raises IPL to IPL$_RESCHED to block this interrupt. For example, the System Generation (SYSGEN) utility raises to this IPL while performing a WRITE ACTIVE command.

This is one of the two IPLs used for synchronization that do not have an associated spinlock.

9.4.9 IPL$_ASTDEL (2)

The IPL of AST interrupts is architecturally defined and cannot be changed by operating system software. Throughout the book, therefore, this IPL is referred to explicitly as 2 rather than symbolically as IPL$_ASTDEL.

IPL 2 execution blocks AST interrupts within a process. When system service procedures raise IPL to 2, they are blocking the delivery of all ASTs, but often particularly a kernel AST that causes process deletion or suspension. In other words, if a process is executing at IPL 2 or above, it cannot be deleted or suspended. It is also possible to block process deletion and suspension by disabling AST delivery to kernel mode.

Raising IPL to 2 prevents process deletion between the time that some system resource (such as system dynamic memory) is allocated and the time that ownership of that resource is recorded (such as the insertion of a data structure into a list). For example, the $QIO system service executes at IPL 2 from the time that an I/O request packet is allocated from nonpaged dynamic memory until that packet is queued to a UCB or placed into the I/O postprocessing queue.

IPL 2 is also significant in that it is the highest IPL at which page faults are permitted. If a page fault occurs above IPL 2, the page fault exception service routine generates the fatal bugcheck PGFIPLHI. If there is any possibility that a page fault can occur, because either the code executing or the data being referenced is pageable, then that code cannot execute above IPL 2. The converse of this constraint is that any code that executes above IPL 2, its linkage section, any routines it invokes, and all data referenced by such code, must be locked into memory in some way. Appendix *Use of Listings and Map Files* shows some of the techniques that the VMS executive uses dynamically to lock into memory code or data referenced from IPLs above 2.

This is one of the two IPLs used for synchronization that do not have an associated spinlock. Because the IPL associated with a spinlock must be at least 3, to block rescheduling, there could not be a spinlock with this IPL.

9.5 Spinlocks

In the VAX/VMS Version 5 implementation of spinlocks, a spinlock is acquired by a processor to synchronize access to data shared by members of an SMP system. The most basic form of spinlock is a bit that describes the state of a particular set of shared data; the bit is set to indicate that a processor is accessing the data. The bit is tested and set, or tested and cleared, atomically with respect to any other threads of execution on the same or other processors. A spinlock enables a set of processors to serialize their access to shared data.

As previously described, spinlock support remains in *censored* VMS, partly for simplicity in porting VAX/VMS Version 5.4 to *censored* VMS Version 1. The descriptions in this and following sections, therefore, are historical and philosophical rather than directly relevant to uniprocessor *censored* VMS operations in this release.

A processor that needs access to some shared data tests and sets the spinlock associated with that data using an interlocked bit test and set instruction. If the bit was clear, the processor is allowed to access the data. This is known as locking or acquiring the spinlock. If the bit was already set, the processor must wait, because another processor is accessing the data.

A waiting processor essentially spins in a tight loop, executing repeated bit test instructions to test the state of the spinlock. This is known as a busy wait. It is from this spinning that the term *spinlock* derives. The busy wait ends when the processor accessing the data clears the bit with an interlocked operation to indicate that it is done. Clearing the bit is known as unlocking or releasing the spinlock.

A resource synchronized through elevated IPL on a uniprocessor is synchronized through a combination of spinlock and elevated IPL on an SMP system. A thread of execution running on one processor acquires a spinlock to serialize access to the data with threads of execution running on other processors. Before acquiring the spinlock, the thread of execution raises IPL to block accesses by other threads of execution running on the same processor. The IPL value is determined by the spinlock being locked.

The concept of spinlock adds a dimension to the concept of raising IPL and extends its effect across all processors in the SMP system. Acquiring a spinlock, however, is different from causing IPL to be raised on all SMP members to block all threads running at lower IPL. Instead, only those threads of execution that try to acquire a spinlock owned by another processor are blocked. This provides more parallelism than simply extending an IPL raise would. Furthermore, since some IPLs, such as IPL 8, are now represented by multiple spinlocks, the granularity of locking is finer, allowing for even more parallelism.

To adapt more easily from IPL-based synchronization to the needs of symmetric multiprocessing, the implementation of spinlocks permits nested acquisitions of a spinlock. For example, many routines that manipulate the scheduler database raised IPL to IPL$_SYNCH in earlier versions of VAX/VMS. If one routine already at IPL$_SYNCH invoked another routine that raised IPL to IPL$_SYNCH to access the same database, no harm was done. On a version of VMS that supports SMP, this sequence results in multiple concurrent, or nested, acquisitions of the SCHED spinlock by the same processor.

A bit used as a spinlock is actually part of a larger data structure called a spinlock control block. Some spinlock control blocks are defined in the VMS executive; these are called static spinlocks. Others, created during system operation, are called dynamic. Section 9.5.1 describes the spinlock control block; Section 9.5.2, static spinlocks; and Section 9.5.3, dynamic spinlocks.

To acquire or release a spinlock, kernel mode code invokes one of several macros, identifying the spinlock in a macro argument. The macros generate code that dispatches to executive routines that perform the actual spinlock operations.

There are potentially three different versions of these routines, conditionally assembled from one source module and built into three loadable executive images. [*Censored*] VMS supplies only a uniprocessor version of this image.

Section 9.5.7 summarizes the spinlock routines.

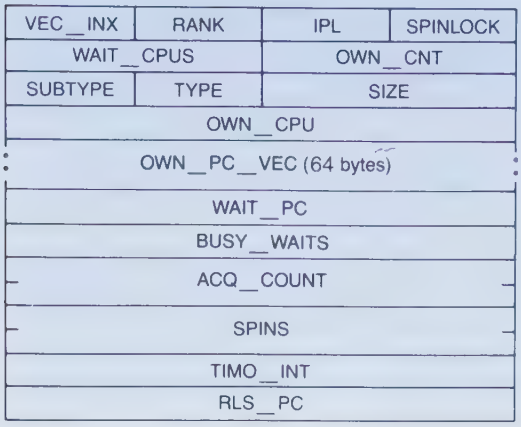
9.5.1 Spinlock Control Block

Figure 9–6 shows the layout of a spinlock control block. The macro $SPLDEF defines symbolic names for its fields.

As implemented for VAX/VMS SMP support, the low bit of field SPL$B_SPINLOCK has two meanings: one for the streamlined spinlock routines, and one for the full-checking routines. The former use the basic form of spinlock; the low bit of SPL$B_SPINLOCK is the actual spinlock. For the full-checking routines, this bit merely serializes access to the spinlock control block; the fields SPL$W_OWN_CNT and SPL$L_OWN_CPU are the actual lock. The distinction is not relevant for a uniprocessor system.

SPL$B_IPL specifies the IPL associated with the spinlock, the minimum IPL at which a processor runs while holding the spinlock.

Figure 9–6 Layout of a Spinlock Control Block



SPL$B_RANK defines the rank of the spinlock. Spinlock rank is stored in an inverted form. Its possible values range from 0 to 31, with 0 being the highest rank. That is, rank increases from 31 to 30 to 29, and so on. This chapter uses the inverted form in its descriptions. Each static spinlock has a unique rank; all dynamic spinlocks have the same rank, which is 31. A thread of execution that acquires multiple static spinlocks must acquire them in increasing rank (see Section 9.5.4).

SPL$W_OWN_CNT records how many concurrent and nested times a processor has locked the spinlock. This field is initialized to –1 to indicate that a spinlock is unowned. With an owner count biased by –1, the acquire code can more easily distinguish increments that cause a transition between unowned and owned from those that do not. When a processor first acquires a spinlock, the value is incremented to 0. If a thread of execution invokes another routine that acquires the same spinlock, the owner count is incremented to 1.

SPL$W_SIZE and SPL$B_TYPE contain the spinlock control block's size and type. SPL$B_SUBTYPE indicates the type of spinlock: static spinlock, fork spinlock, or device spinlock. These types are described further in the sections that follow.

SPL$L_OWN_CPU contains the address of the per-CPU database of the processor that has acquired the spinlock. The address is recorded when a processor acquires the spinlock. The field is cleared when a processor releases its last nested acquisition of the lock.

SPL$L_TIMO_INT is the maximum amount of time a processor can wait for the spinlock. After this interval has elapsed, the attempted spinlock acquisition times out. During system initialization, the timeout value is initialized to one of two values: if the spinlock IPL is less than or equal to 8, the value of the SYSGEN parameter SMP_LNGSPINWAIT is used; otherwise, the value of the SYSGEN parameter SMP_SPINWAIT is used. There are two different values because the MMG and SCHED spinlocks are occasionally held longer than would be reasonable for spinlocks with higher IPLs.

The spinlock control block fields that follow are used only by the full-checking version of the spinlock routines.

SPL$W_WAIT_CPUS contains the number of processors waiting to acquire the spinlock.

The 16 longwords beginning at SPL$L_OWN_PC_VEC form a ring buffer that records the most recent PCs from which an owner CPU acquired and released the spinlock. SPL$B_VEC_INX contains the index of the next entry to be written in the ring buffer.

SPL$L_WAIT_PC contains the address of the most recent busy wait for the spinlock.

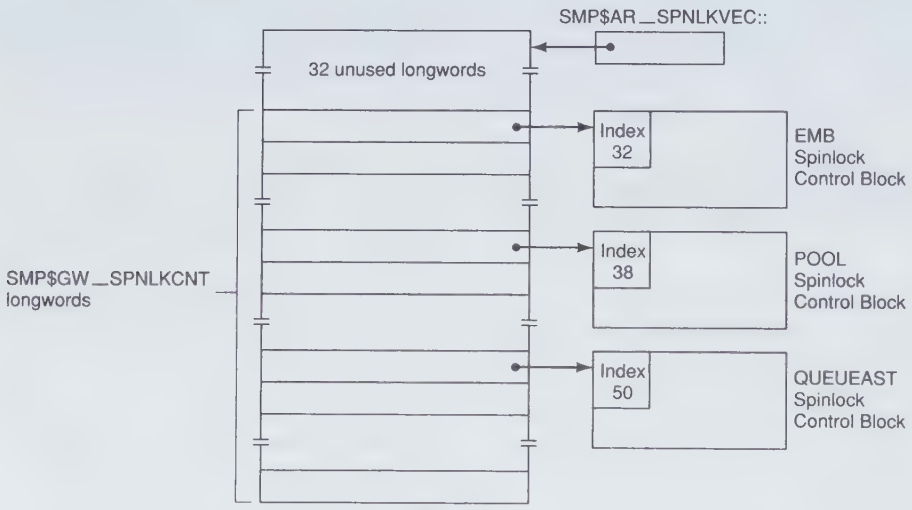
SPL$Q_ACQ_COUNT is the cumulative number of successful acquisitions of the spinlock. SPL$L_BUSY_WAITS is the cumulative number of failed acquisitions. It is incremented once when a processor fails to acquire a spinlock on its first attempt. SPL$Q_SPINS is the cumulative number of spins. It is incremented when a processor fails to acquire a spinlock on its first attempt and again at each unsuccessful attempt during the processor's spinwait.

SPL$L_RLS_PC is the most recent return PC of a thread of execution that releases all nested acquisitions at once.

9.5.2 Static Spinlocks

All static spinlock control blocks are defined in module LDAT, which also contains a table listing their addresses. The global SMP$AR_SPNLKVEC contains the address of the table, and SMP$GW_SPNLKCNT contains the number of spinlocks in the table. Figure 9–7 shows this table and several representative spinlocks.

Figure 9–7 Static Spinlock Table



A static spinlock is identified by the position of its address in the table. This is known as its index, and is the longword postindex offset of its address in the table. The macro $SPLCODDEF defines symbolic names for these indexes; for example, SPL$C_SCHED is the index of the SCHED spinlock. The lowest index is 32. Table entries with lower indexes are empty. Having index values of 32 or greater makes it easy to distinguish a spinlock index from an IPL value. (When spinlocks were first introduced with VAX/VMS Version 5, it was necessary to make this distinction: some fork blocks contained an IPL and others a fork spinlock index. In [*censored*] VMS, all fork blocks contain a fork spinlock index.)

Each static spinlock is aligned on a 128-byte boundary to prevent unnecessary failures accessing the low bit of SPL$B_SPINLOCK. This bit is tested and set, or tested and cleared, using the load-locked/store-conditional mechanism described in Section 9.2.2. If two spinlocks were located in the same lock range, writing to fields in one could cause a concurrent store-conditional attempt in the other to fail.

Table 9–2 lists the static spinlocks with a brief description of what each synchronizes and its associated IPL. The spinlocks are listed in order by rank, with lower ranking spinlocks first. Section 9.5.5 describes their use in more detail.

A static spinlock that synchronizes fork processing is called a fork spinlock, often shortened to fork lock. A device UCB and any other type of fork block (FKB) identify the driver's fork lock, and indirectly its fork IPL, by specifying the spinlock index in the field UCB$B_FLCK or FKB$B_FLCK. Some static spinlocks are never used as fork locks; their associated IPLs are not in the fork IPL range. Some static spinlocks are only used as fork locks, for example, the IOLOCK8 spinlock, and some are sometimes used as fork locks, for example, the MAILBOX spinlock.

The SPL$B_SUBTYPE field of a spinlock used only as a fork lock contains the value SPL$C_SPL_FORKLOCK; all other static spinlocks are identified simply as spinlocks, with the value SPL$C_SPL_SPINLOCK. The main distinction between fork locks and other static spinlocks is that fork locks are typically acquired and released through different macros. Kernel mode code acquires and releases a static spinlock by invoking the LOCK and UNLOCK macros; the FORKLOCK and FORKUNLOCK macros are used for fork locks. Section 9.5.6 describes these macros.

Another table with information about static spinlocks is the spinlock IPL table, at global symbol SMP$AL_IPLVEC. The table is indexed by static spinlock index and contains the IPL corresponding to that spinlock. This table is referenced by the routine EXE$PRIMITIVE_FORK (see Chapter *Software Interrupts)*.

Table 9–2 Static Spinlocks

Name	IPL	Synchronizes
QUEUEAST	6	IPL 6 fork processing
FILSYS	8	File system data structures such as file control blocks
IOLOCK8/SCS[1]	8	IPL 8 fork processing; SCS-related code
TIMER	8	Timer queue entries
JIB	8	Job information block fields JIB$L_BYTCNT and JIB$L_BYTLM
IO_MISC	8	Controller register access mailbox allocation and deallocation
MMG	8	Memory management data structures
SCHED	8	Scheduler database
IOLOCK9	9	IPL 9 fork processing
IOLOCK10	10	IPL 10 fork processing
IOLOCK11	11	IPL 11 fork processing
MAILBOX	11	Writing mailbox messages
POOL	11	Nonpaged pool lists and related data
PERFMON	15	Performance monitoring
INVALIDATE	21	Translation buffer invalidation
HWCLK	22	Hardware clock database
MEGA	31	Miscellaneous data structures such as the fork-and-wait queue
MCHECK/EMB[1]	31	Machine check serialization; error log buffers

[1]These two names are synonyms for the same spinlock.

9.5.3 Dynamic Spinlocks

A dynamic spinlock is not listed in the spinlock table and has no index. A dynamic spinlock control block is allocated from nonpaged pool and is identified by its address. Currently, the only type of dynamic spinlock VMS uses is a device spinlock, usually called a device lock. The SPL$B_SUBTYPE field of a device lock's spinlock control block contains the value SPL$C_SPL_DEVICELOCK.

All dynamic spinlocks have the same rank, 31. However, the field SPL$B_RANK in a dynamic spinlock control block is initialized to –1 for quick identification in the routines that acquire and release spinlocks.

As the [IOGEN]SYS$LOAD_DRIVER procedure configures devices, it creates device locks. It invokes the routine SMP$ALLOC_SPL to create a device lock and SMP$INIT_SPL to initialize it. Both routines are in module SPINLOCKS.

In general, there is one device lock for each unique device controller. SYS$LOAD_DRIVER stores its address in the controller's channel request block field CRB$L_DLCK and in the field UCB$L_DLCK for each unit on that controller.

A device driver acquires and releases the device lock by invoking the DEVICELOCK and DEVICEUNLOCK macros (see Section 9.5.6). The device lock synchronizes access to the controller's registers and to fields in the UCB that describe the controller's state.

9.5.4 Rules for Acquiring and Releasing Spinlocks

For synchronization with spinlocks to be successful, threads of execution that use spinlocks must follow certain rules.

All threads of execution that access shared data must first acquire the spinlock associated with it. A thread of execution that acquires a spinlock to serialize access to shared data is guaranteed exclusive access to the data while it holds the spinlock. Thus, all its modifications to the data can be considered atomic with respect to another thread trying to acquire the same spinlock to access the same data. To ensure this degree of atomicity, the implementation of spinlocks does not include breaking spinlock deadlocks. Rather, deadlocks are prevented by requiring threads of execution that use spinlocks to acquire spinlocks in a particular order.

The rank values of static spinlocks reflect VMS code paths and interdependence among the shared data structures protected by spinlocks. A thread of execution that acquires multiple spinlocks must acquire them in order by increasing rank. This rule prevents a deadlock such as the following: One processor has acquired spinlock A and is busy waiting to acquire spinlock B to complete its task; a second processor has acquired spinlock B and is busy waiting to acquire spinlock A to complete its task.

All device locks share the same rank, 31, which is lower than that of any static spinlock. However, a processor holding a static spinlock may acquire a device lock; the rule previously listed does not apply to acquisition of a device lock. The assumption is that the shared resource protected by a device lock is not dependent on the resources protected by the static spinlocks. Furthermore, each device lock is assumed to be independent of others, and a processor is permitted to hold more than one device lock at a time. All code acquiring multiple device locks concurrently must be written to prevent deadlocks; all threads must acquire such device locks in the same order.

A thread of execution about to acquire a spinlock must be running at an IPL less than or equal to that of the spinlock. This is analogous to the principle of raising IPL to synchronize on a uniprocessor system. This rule prevents the following type of synchronization failure:

1. Thread A, running at IPL x, acquires a spinlock and begins to manipulate the database it protects.
2. An interrupt at IPL $x + 1$ is requested and granted on the same processor, and thread B begins execution, interrupting thread A.
3. To access the same database, thread B tries to acquire its spinlock. Because threads A and B are running on the same processor, the nested acquisition is successful, and thread B begins to manipulate the database left in an inconsistent state by the interruption to thread A.

A thread of execution that has acquired a spinlock may raise IPL but must not lower it below the value associated with the spinlock. Lowering IPL could lead to the synchronization failure just described.

9.5.5 Use of Static Spinlocks

The sections that follow describe the use of each of the static spinlocks.

9.5.5.1 Use of the QUEUEAST Spinlock

The QUEUEAST spinlock synchronizes fork processing at IPL 6. The need for IPL 6 fork processing is largely historical, based on constraints from VAX/VMS Version 3 and earlier versions, as described in Section 9.4.7. IPL 6 is also used for fork processing that is lower priority than I/O fork processing and can be deferred. For example, the control block and stack associated with a deleted kernel process (see Chapter *Software Interrupts)* are deallocated within an IPL 6 fork process.

9.5.5.2 Use of the FILSYS Spinlock

The file system database consists of data structures that describe the mount state of a volume and the condition of open files on the volume. The FILSYS spinlock synchronizes access to pieces of the file system database that are accessed by routines external to the file system. (As described in Appendix *Lock and Resource Use by VMS Components,* lock management system services synchronize access to much of the file system database.)

For example, each open file is described by one or more window control blocks (WCBs). A WCB contains retrieval pointers that map the virtual blocks of a file to logical blocks on a device. As part of processing an I/O request to a file, IOC$MAPVBLK, in module IOSUBRAMS, uses WCB contents to convert virtual block numbers to their equivalent logical block numbers. IOC$MAPVBLK and the file system routines that alter WCBs synchronize their access to WCBs by acquiring the FILSYS spinlock.

9.5.5.3 Use of the IOLOCK8/SCS Spinlock

A driver can specify one of the IOLOCK*x* spinlocks as its fork lock. A device UCB, which is also used as a fork block, contains the fork spinlock index in the field UCB$B_FLCK. The fork lock synchronizes access to data structures modified by the fork process, in particular, its UCB.

To synchronize access to UCB fields manipulated at fork level, an executive routine or the driver fork process itself acquires the spinlock specified in UCB$B_FLCK.

IOLOCK8 is the fork lock most commonly used by device driver fork processes. It is used by all standard drivers that compete for shared resources like a device controller.

IOLOCK8 and SCS are two names for the same spinlock. SCS routines and lock manager routines coordinate access to SCS and lock management data structures using the SCS spinlock. An SCS routine that executes as a fork process uses the SCS spinlock as its fork lock. The $ENQ and $DEQ system services acquire the SCS spinlock before altering the lock database.

9.5.5.4 Use of the TIMER Spinlock

The software timer interrupt service routine holds the TIMER and HWCLK spinlocks while it tests whether the first entry in the time-ordered queue of TQEs has expired.

The routines that insert and remove TQEs from the timer queue hold the TIMER spinlock while they manipulate the queue. The Set Time ($SETIME) system service acquires the TIMER spinlock when it resets the system time and reorders the timer queue as a result of recalibrating pending TQEs with delta times. Chapter *Time Support* provides more information on these two interrupt service routines and the timer queue.

9.5.5.5 Use of the JIB Spinlock

The JIB spinlock synchronizes access to the job information block (JIB) fields JIB$L_BYTCNT and JIB$L_BYTLM. The process context routines EXE$DEBIT_BYTCNT and EXE$CREDIT_BYTCNT and their alternative entry points acquire this spinlock to debit and credit the job's available byte count. The intent of the spinlock is to block other processes in the same job from simultaneously accessing these fields. Because there is only one systemwide JIB spinlock, however, all other processes in the system are blocked from accessing these fields in their own JIBs. This implementation, however, has the virtue of simplicity, and such accesses are believed to be sufficiently infrequent so as to present no performance issue.

9.5.5.6 Use of the IO_MISC Spinlock

The IO_MISC spinlock synchronizes access to the data structures that describe the collection of controller register access mailbox (CRAM) data structures but does not synchronize access to each individual CRAM. Prior to accessing a controller register, a device driver allocates and initializes a CRAM with the address of a particular I/O space register. The routines that a driver calls to allocate and deallocate a CRAM acquire this spinlock. Chapters *Overview of the I/O Subsystem* and *I/O Processing* describe accessing controller registers through hardware mailboxes.

9.5.5.7 Use of the MMG Spinlock

The MMG spinlock synchronizes access to the memory management database. This includes the page frame number database, section tables, page and swap file bitmaps, list of available system page table entries, and working set lists.

Its main users are the page fault exception service routine, the swapper, memory management system services, and routines that lock and unlock direct I/O buffer pages into memory.

9.5.5.8 Use of the SCHED Spinlock

The SCHED spinlock synchronizes access to the scheduler database, the set of software process control blocks and their state queues, and mutex data structures. It also synchronizes access to a process's AST data, such as the PCB$B_ASTACT bits and the queues of pending AST control blocks.

9.5.5.9 Use of the IOLOCK*n* Fork Spinlocks

The spinlocks IOLOCK9, IOLOCK10, and IOLOCK11 are fork locks intended for use on any processor. Their use is similar to that of IOLOCK8, although they are not as commonly used.

9.5.5.10 Use of the MAILBOX Spinlock

The MAILBOX spinlock synchronizes access to mailboxes. It is the fork lock for the mailbox driver. The mailbox driver's internal routines EXE$SNDEVMSG and EXE$WRTMAILBOX, invoked by device drivers to write messages to a mailbox, acquire this spinlock to synchronize access to the mailbox.

9.5.5.11 Use of the POOL Spinlock

The POOL spinlock synchronizes access to the nonpaged variable-length list. Its main users are routines such as EXE$ALONONPAGED and EXE$DEANONPAGED, in module MEMORYALC. It also synchronizes access to the performance monitoring statistics kept on nonpaged variable-length pool allocation failures. Note that because the nonpaged lookaside lists are accessed with instructions for which the processor provides synchronization, they may be accessed without the POOL spinlock.

The code that implements the DCL SHOW MEMORY command acquires the POOL spinlock while it scans the list to collect information for its display. The Monitor utility acquires the POOL spinlock while it gathers information on pool use.

9.5.5.12 Use of the PERFMON Spinlock

The PERFMON spinlock synchronizes access to the I/O performance database. Its main users are routines such as PMS$START_REQ, in module IOPERFORM.

9.5.5.13 Use of the INVALIDATE Spinlock

The routine MMG$TBI_SINGLE, in module PAGEFAULT, is invoked to flush a stale entry from the translation buffer whenever a valid page table entry is made invalid. If the address is a system space address in an SMP system, multiple processors might have the translation cached. To invalidate a cached system space address translation, a member of an SMP system would acquire the INVALIDATE spinlock. The spinlock would prevent more than one processor at a time from initiating the sequence required for all SMP members to invalidate the entry in their translation buffers.

9.5.5.14 Use of the HWCLK Spinlock

The HWCLK spinlock synchronizes access to the cell EXE$GQ_1ST_TIME, which contains the expiration time of the first TQE in the list.

The interval timer interrupt service routine acquires the HWCLK spinlock to test whether the first TQE has expired. If it has, the interrupt service routine requests an IPL 7 interrupt for the software timer. The software timer interrupt service routine acquires the TIMER and HWCLK spinlocks while it tests whether the first TQE has expired.

The routines that insert and remove TQEs from the timer queue acquire the HWCLK spinlock if they have to manipulate the first TQE in the list. Chapter *Time Support* gives more information on these two interrupt service routines and the timer queue.

9.5.5.15 Use of the MEGA Spinlock

The MEGA spinlock synchronizes access to the fork and wait queue, used by fork processes to stall themselves for approximately half a second (see Chapter *Software Interrupts*).

9.5.5.16 Use of the MCHECK Spinlock

The machine check exception service routines for CPUs that can be members of an SMP system acquire the MCHECK spinlock as needed. On some systems, a single hardware error can cause concurrent machine checks on all CPUs; the spinlock could be used to enable only one processor at a time to read or write memory controller interface registers. Other CPU-specific error interrupt service routines acquire the spinlock for similar reasons.

9.5.5.17 Use of the EMB Spinlock

The EMB spinlock synchronizes access to the error log allocation buffers (see Chapter *Error Handling).* The routines that reserve and release pieces of error log allocation buffer for error messages acquire the EMB spinlock.

The ERRFMT process acquires the EMB spinlock when it is altering data structures that describe the state of the error log allocation buffer. As Chapter *Error Handling* describes, ERRFMT copies an error log allocation buffer in several stages. It examines the error log buffer status flags and message counts with the spinlock held. If it can copy the buffer, it sets a flag in the buffer to inhibit further allocations in it and then releases the spinlock. At IPL 0, ERRFMT copies the error log allocation buffer to its P0 space, and formats and writes the messages to the error log file.

9.5.6 Macros for Acquiring and Releasing Spinlocks

There are three sets of VAX MACRO macros for acquiring and releasing spinlocks:

- LOCK and UNLOCK for static spinlocks
- FORKLOCK and FORKUNLOCK for static spinlocks used to synchronize fork processing
- DEVICELOCK and DEVICEUNLOCK for dynamic spinlocks

These macros facilitate writing code that can synchronize properly whether it executes on a uniprocessor or a member of an SMP system. Each of these macros has a number of arguments, only a few of which are described here.

These macros differ primarily in the way their arguments identify the spinlock of interest:

- An argument to LOCK and UNLOCK specifies the symbolic index of a static spinlock.
- In a typical use of FORKLOCK or FORKUNLOCK, R5 contains the address of a UCB in which the field UCB$B_FLCK has a static spinlock index.
- In a typical use of DEVICELOCK or DEVICEUNLOCK, R5 contains the address of a UCB in which the field UCB$L_DLCK has the address of the device lock.

The lock macros generate the following approximate sequence:

1. Optionally (determined by macro argument SAVIPL), save the current IPL.
2. If SMP is not enabled, set IPL as requested and branch around the rest of the instructions. The low bit of global SMP$GL_FLAGS is set when SMP is enabled.
3. Optionally (determined by macro argument PRESERVE, whose default value is the string YES), save all 64 bits of R0.
4. Store the static spinlock index or the address of a dynamic spinlock in R0.
5. Invoke SMP$ACQNOIPL in the case of a static spinlock with macro argument CONDITION=NOSETIPL; SMP$ACQUIRE in any other case of a static spinlock; or SMP$ACQUIREL or SMP$ACQNOIPL in the case of a dynamic spinlock.
6. If R0 was saved, restore it.

A sample invocation of LOCK with its expansion follows:

```
;The macro invocation
;locks spinlock with index SPL$C_MMG
        LOCK    LOCKNAME=MMG,-      ;Lock MMG database
                PRESERVE=NO         ;Don't preserve R0

;Its expansion
        ASSUME  SMP$V_ENABLED EQ 0
        BLBC    SMP$GL_FLAGS,30002$
        MOVL    S^#SPL$C_MMG,R0
        JSB     G^SMP$ACQUIRE
        BRB     30003$
30002$: MTPR    #IPL$_MMG,#PR$_IPL
30003$:
```

A sample invocation of FORKLOCK with its expansion follows:

```
;The macro invocation
;locks spinlock whose index is in UCB$B_FLCK
        FORKLOCK -
                UCB$B_FLCK(R5),-    ;Lock fork access
                SAVIPL=-(SP)        ;Save current IPL

;Its expansion
        MFPR    S^#PR$_IPL,-(SP)
        $PUSH64 R0
        MOVZBL  UCB$B_FLCK(R5),R0
        ASSUME  SMP$V_ENABLED EQ 0
        BLBC    SMP$GL_FLAGS,30002$
        JSB     G^SMP$ACQUIRE
        BRB     30003$
30002$: MTPR    SMP$AL_IPLVEC[R0],S^#PR$_IPL
30003$: $POP64  R0
```

A sample invocation of DEVICELOCK with its expansion follows:

```
;The macro invocation
;locks spinlock whose address is in UCB$L_DLCK
        DEVICELOCK -
                LOCKADDR=UCB$L_DLCK(R5),- ;Lock device interrupts
                CONDITION=NOSETIPL,-      ;Don't alter IPL
                PRESERVE=NO               ;Don't preserve R0

;Its expansion
        ASSUME  SMP$V_ENABLED EQ 0
        BLBC    SMP$GL_FLAGS,30002$
        MOVL    UCB$L_DLCK(R5),R0
        JSB     G^SMP$ACQNOIPL
30002$:
```

The unlock macros generate the following approximate code sequence:

1. If SMP is not enabled, go to step 6.
2. Optionally (determined by macro argument PRESERVE, whose default value is the string YES), save all 64 bits of R0.
3. Store the static spinlock index or the address of a dynamic spinlock in R0.
4. If the macro argument CONDITION=RESTORE is present, invoke SMP$RESTORE to relinquish one acquisition of a static spinlock or SMP$RESTOREL for a dynamic spinlock.

   If the macro argument is not present, invoke SMP$RELEASE to relinquish all nested acquisitions of a static spinlock or SMP$RELEASEL for a dynamic spinlock.
5. If R0 was saved, restore it.
6. Optionally (determined by macro argument NEWIPL), set IPL as requested.

A sample invocation of UNLOCK with its expansion follows:

```
;The macro invocation
        UNLOCK   LOCKNAME=INVALIDATE,-
                 PRESERVE=NO,-      ;Don't save R0
                 NEWIPL=(SP)+       ;Restore IPL from stack

;Its expansion
        ASSUME  SMP$V_ENABLED EQ 0
        BLBC    SMP$GL_FLAGS,30002$
        MOVL    S^#SPL$C_INVALIDATE,R0
        JSB     G^SMP$RELEASE
30002$: MTPR    (SP)+,S^#PR$_IPL
```

A sample invocation of FORKUNLOCK with its expansion follows:

```
;The macro invocation
        FORKUNLOCK -
                UCB$B_FLCK(R5),-  ;Release fork access
                NEWIPL=(SP)+      ;Restore IPL from stack

;Its expansion
        ASSUME  SMP$V_ENABLED EQ 0
        BLBC    SMP$GL_FLAGS,30002$
        $PUSH64 R0
        MOVZBL  UCB$B_FLCK(R5),R0
        JSB     G^SMP$RELEASE
        $POP64  R0
30002$: MTPR    (SP)+,S^#PR$_IPL
```

A sample invocation of DEVICEUNLOCK with its expansion follows. This example results in dispatch to SMP$RESTOREL rather than to SMP$RELEASEL.

```
;The macro invocation
        DEVICEUNLOCK -
                LOCKADDR=UCB$L_DLCK(R5),-
                                -;Release device interrupts
                NEWIPL=(SP)+,-    ;Restore IPL
                CONDITION=RESTORE ;Conditionally release spinlock

;Its expansion
        ASSUME  SMP$V_ENABLED EQ 0
        BLBC    SMP$GL_FLAGS,30002$
        $PUSH64 R0
        MOVL    UCB$L_DLCK(R5),R0
        JSB     G^SMP$RESTOREL
        $POP64  R0
30002$: MTPR   (SP)+,S^#PR$_IPL
```

9.5.7 Spinlock Routines

As described in Section 9.5, there are potentially three versions of the spinlock routines, conditionally assembled from one source. Having different versions of the routines hides the details of the actual synchronization method used. It enables code requiring synchronization to contain hooks suitable for SMP, should [*censored*] VMS support such a configuration. It also minimizes the changes necessary in code ported from VAX/VMS. This section summarizes the spinlock routines.

The following spinlock routines run in kernel mode at IPL 3 and above:

- SMP$ACQUIRE—Acquire a static spinlock
- SMP$ACQUIREL—Acquire a dynamic spinlock
- SMP$ACQNOIPL—Acquire a dynamic spinlock without altering IPL
- SMP$RESTORE—Relinquish one acquisition of a static spinlock
- SMP$RESTOREL—Relinquish one acquisition of a dynamic spinlock
- SMP$RELEASE—Relinquish all nested acquisitions of a static spinlock
- SMP$RELEASEL—Relinquish all nested acquisitions of a dynamic spinlock

The spinlock lock macros dispatch to SMP$ACQUIRE or to one of its alternative entry points, SMP$ACQNOIPL or SMP$ACQUIREL.

The spinlock unlock macro invocations that request a restore (relinquish one acquisition of a spinlock) dispatch to SMP$RESTORE or to its alternative entry point SMP$RESTOREL.

Because the macros all test whether SMP is enabled, it is unlikely that control would ever be transferred to the uniprocessor versions of the spinlock routines, in module SPINLOCKS_UNI.

9.6 Serialized Access

VMS combines software interrupts and queues to serialize several requests for the same data structure or procedure. An example of this serialization is the use of fork processes by device drivers and other parts of the executive.

Fork processing is the technique that allows a device driver to lower IPL in a manner consistent with the interrupt nesting scheme defined by the *censored* architecture. When a device driver receives control in response to a device interrupt, it performs whatever steps are necessary to service the interrupt at device IPL. For example, before dismissing the device interrupt, it reads any device registers whose contents would be destroyed by another interrupt.

Usually, some processing can be deferred. For DMA devices, an interrupt signifies either completion of the operation or an error. The code that distinguishes these two cases and performs error processing is usually lengthy. If it executed at device IPL for extended periods of time, it would reduce response to high-priority interrupts.

To delay further processing until IPL drops below the fork IPL associated with this driver, the device driver interrupt service code invokes an executive routine, EXE$PRIMITIVE_FORK, in module FORKCNTRL. EXE$PRIMITIVE_FORK saves some minimal context in a fork block. It saves two general registers and the procedure value of the routine in the driver to which control should return when IPL drops.

EXE$PRIMITIVE_FORK extracts the fork block field FKB$B_FLCK, which contains a static spinlock index, and indexes the static spinlock IPL table with it to obtain the IPL value associated with that spinlock. The routine inserts the fork block at the end of the fork queue for that IPL and requests a software interrupt at that IPL if the queue was previously empty.

A thread of execution that creates a fork process can use any appropriate static spinlock as its fork lock. The only requirement is that the spinlock IPL be one at which fork processing is performed: 6, 8, 9, 10, or 11.

Chapter *Software Interrupts* describes fork processing in further detail.

9.7 Mutual Exclusion Semaphores (Mutexes)

The synchronization techniques described so far all execute at elevated IPL, thus blocking certain operations, such as a rescheduling request. However, in some situations requiring synchronization, elevated IPL is an unacceptable technique. One reason elevated IPL might be unacceptable is that the processor would have to remain at an elevated IPL for an indeterminately long time because of the structure of the data. For example, associating to a common event block cluster requires a search of the list of common event blocks (CEBs) for the specified CEB. This might be a lengthy operation on a system with many CEBs.

Furthermore, elevated IPL is unacceptable for synchronizing access to pageable data. VMS bugchecks if a page fault occurs at an IPL above 2. Thus, a pageable data structure cannot be protected by elevating IPL.

One synchronization mechanism that does not require elevated IPL is a mutual exclusion semaphore, or mutex. VMS uses mutexes for synchronizing kernel mode accesses to certain shared data structures. A mutex is essentially a counter that controls read or write access to a given data structure or database. VMS allows either multiple readers or one writer of a data structure or database synchronized through mutex acquisition. Typically, the threads of execution whose accesses are synchronized through a mutex are process context threads.

Access to a mutex itself must be gained at elevated IPL with the SCHED spinlock held. However, once a mutex is acquired, elevated IPL is not required to access the database represented by the mutex.

Table 9–3 lists the executive data structures protected by mutexes and the names of the corresponding mutexes.

A mutex is a data structure consisting of a single longword. Figure 9–8 shows its layout. The macro $MTXDEF defines symbolic names for its fields. Its low-order word, field MTX$W_OWNCNT, contains the number of processes accessing the data, that is, the number of processes that have locked the mutex. The owner count value is initialized to –1 to indicate no owners. Thus, a mutex with a 0 in the low-order word has one owner. Biasing the owner count by –1 simplifies the code that tests for the transition between unowned and owned. The high-order word of a mutex, field MTX$W_STS, contains status flags. The only flag currently implemented, MTX$V_WRT, is set to indicate that a write is either in progress or pending for this mutex.

The process control block (PCB) field PCB$W_MTXCNT contains the number of mutexes a process currently owns. This field is initialized to zero and incremented each time a process acquires a mutex.

Typically, executive code invokes the routines described in the following sections to lock or unlock a mutex whose address is specified in R0. Some subsystems, however, use routines such as SCH$IOLOCKR, SCH$IOLOCKW, and SCH$IOUNLOCK, which load the address of the I/O database mutex into R0 and transfer to their more general counterparts.

Figure 9–8 Layout of a Mutex

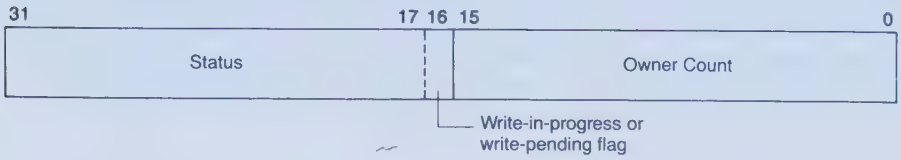


Table 9–3 List of Data Structures Protected by Mutexes

Data Structure	Global Name of Mutex
Shared logical name data structures	LNM$AL_MUTEX
I/O database	IOC$GL_MUTEX[1]
Common event block list	EXE$GL_CEBMTX
Paged dynamic memory list	EXE$GL_PGDYNMTX
Global section descriptor list	EXE$GL_GSDMTX
Not currently used	EXE$GL_SHMGSMTX
Not currently used	EXE$GL_SHMMBMTX
Not currently used	EXE$GL_ENQMTX
Line printer unit control block	UCB$L_LP_MUTEX[2]
Not currently used	EXE$GL_ACLMTX
System intruder lists	CIA$GL_MUTEX
Object rights block access control list	ORB$L_ACL_MUTEX[3]
System service database	CHANGE_MODE_MUTEX[4]
Terminal fallback database	TFF$L_VEC_MUTEX[5]
Loadable executive image data structures	EXE$GL_BASIMGMTX
Not currently used	QMAN$GL_MUTEX

[1]This mutex is used by the Assign Channel and Allocate Device system services when searching through the linked list of device data blocks and UCBs for a device. It is also used when UCBs are added or deleted, for example, during the creation of mailboxes and network devices.

[2]This mutex does not have a fixed address. As a field in a line printer UCB, its location depends on that of the UCB.

[3]This mutex does not have a fixed address. As a field in an ORB, its location depends on that of the ORB.

[4]This mutex is local to the EXCEPTION.EXE loadable executive image and does not have a fixed address.

[5]This mutex does not have a fixed address. As a field in the fallback driver, its location depends on that of the driver.

9.7.1 Locking a Mutex for Read Access

When a process needs read access to a data structure protected by a mutex, it invokes routine SCH$LOCKR, in module MUTEX, with the address of the mutex. SCH$LOCKR takes the following steps:

1. It acquires the SCHED spinlock, raising IPL to IPL$_SCHED.

2. It tests whether the mutex's write flag is set. If so, no further readers are allowed to acquire the mutex. SCH$LOCKR transparently stalls the process (see Section 9.7.3) until the mutex is available.
3. If the write flag is clear, indicating that no write operation is in progress or pending, SCH$LOCKR grants the process read ownership of the mutex—it increments the mutex's owner count and increments the count of mutexes owned by this process.
4. If this mutex is the first that the process currently has locked and if the process is not a real-time process, its current and base priorities are saved in the PCB fields PCB$B_PRISAV and PCB$B_PRIBSAV and then both are elevated to 16. The process receives a priority boost to minimize the time during which it holds the mutex and blocks other processes that require the mutex. To alter the current priority, SCH$LOCKR invokes SCH$CHANGE_CUR_PRIORITY, in module RSE, which is described in Chapter *Scheduling*. The check on the number of owned mutexes prevents a process that gains ownership of two or more mutexes from receiving a permanent priority elevation to 16.
5. SCH$LOCKR releases the SCHED spinlock and returns control to its invoker with IPL at 2.

The process is expected to remain at IPL 2 or above while it owns the mutex to prevent its own deletion or suspension. Neither the Delete Process ($DELPRC) system service nor the Suspend Process ($SUSPND) system service checks whether the target process owns any mutexes. If the process deletion or suspension were to succeed, the locked mutex would no longer be lockable and thus the locked data structure would be inaccessible.

9.7.2 Locking a Mutex for Write Access

When a process needs write access to a data structure that is protected by a mutex, it invokes routine SCH$LOCKW, in module MUTEX, with the address of the mutex. SCH$LOCKW takes the following steps:

1. It acquires the SCHED spinlock, raising IPL to IPL$_SCHED.
2. It tests and sets the mutex's write flag.
3. If the flag was set, no further readers or writers are allowed to acquire the mutex. SCH$LOCKW transparently stalls the process (see Section 9.7.3) until the mutex is available.
4. If the write flag was clear, SCH$LOCKW tests whether there are any current owners of the mutex. If there are, it transparently stalls the process.
5. If the write flag was clear and there are no owners of the mutex, SCH$LOCKW grants the process write ownership of the mutex: it increments MTX$W_OWNCNT and PCB$W_MTXCNT, and it can alter the process's software priority, as previously described. It releases the SCHED spinlock and returns to its invoker at IPL 2.

When SCH$LOCKW stalls the process, the mutex write flag is set so that future requests for read access will also be denied. This prevents a stream of read accesses from continuously locking the mutex. When the last current owner of the mutex releases it, the write flag is cleared. At that point, the highest priority process waiting for the mutex gets first access to it, whether the process is requesting a read or a write access.

If a reader acquires the mutex, other previously waiting would-be readers whose priority is greater than that of the highest priority would-be writer can also acquire read access, as a result of standard scheduling operations. The higher priority would-be readers execute first, and their read accesses are granted. If readers still own the mutex when the would-be writer executes, its attempted write access is blocked again.

An alternative entry point, SCH$LOCKWNOWAIT, returns control to the invoker with R0<0> cleared to indicate failure if the requested mutex is already owned.

9.7.3 Mutex Wait State

SCH$LOCKR and SCH$LOCKW transparently stall a process when its requested mutex acquisition cannot be granted. They store the address of the mutex being requested in the software PCB field PCB$L_EFWM. Because the process is not waiting for an event flag, the field is available for this purpose. They place the process into the miscellaneous wait state (MWAIT). They call the routine that selects a new process to place into execution and that releases the SCHED spinlock. Chapter *Scheduling* describes miscellaneous waits and rescheduling in more detail.

The saved PC of such a process is an address within EXE$KERNEL_WAIT_PS, in module MUTEX. On the stack is a return address within either SCH$LOCKR or SCH$LOCKW, depending on whether the intended mutex acquisition is for read or write. The process's saved PS has a current mode of kernel and IPL 2. When the mutex becomes available, the process becomes computable again. When the process is placed into execution, it reattempts its mutex acquisition.

9.7.4 Unlocking a Mutex

A process releases a mutex by invoking routine SCH$UNLOCK, in module MUTEX, with the address of the mutex to be released. SCH$UNLOCK takes the following steps:

1. It acquires the SCHED spinlock, raising IPL to IPL$_SCHED.
2. It decrements the process's PCB$W_MTXCNT. If this process does not own any more mutexes, SCH$UNLOCK restores the saved base priority from PCB$B_PRIBSAV and invokes SCH$CHANGE_CUR_PRIORITY to restore the current priority from PCB$B_PRISAV.

   If there is a computable resident process with a higher priority than this process's restored priority, SCH$CHANGE_CUR_PRIORITY requests a rescheduling interrupt. This situation is known as delayed preemption of the current process.

3. SCH$UNLOCK also decrements MTX$W_OWNCNT. If the mutex owner count is greater than –1, there are other outstanding owners of this mutex; SCH$UNLOCK simply releases the SCHED spinlock, restoring the IPL at entry, and returns to its invoker.
4. If the mutex count is decremented to –1, the mutex is now unowned. SCH$UNLOCK tests and clears its write flag. If the bit was clear, SCH$UNLOCK releases the SCHED spinlock, restoring the IPL at entry, and returns to its invoker.
5. If the bit was set, there may be processes waiting to acquire this mutex. (A waiting or owning writer would have set this bit, blocking any new potential readers and any writers.) SCH$UNLOCK calls EXE$WAIT_QUEUE_RELEASE, in module MUTEX.

   EXE$WAIT_QUEUE_RELEASE scans the miscellaneous resource wait queue to locate any process whose PCB$L_EFWM field contains the address of the unlocked mutex. For each such process, it reports the availability of the mutex by invoking a scheduler routine to make the process computable. If the priority of any of these processes allows it to preempt a current process, a rescheduling interrupt is requested.

   SCH$UNLOCK then releases the SCHED spinlock, restoring the IPL at entry, and returns to its invoker.

9.7.5 Accessing a Mutex from System Context

Although mutexes were originally designed for use from process context, VMS also allows a system thread of execution to acquire a mutex. This enables a system thread to synchronize its access with those of full processes to a database protected by a mutex. In general, this capability is limited to nonpageable databases, since VMS bugchecks in response to page faults occurring above IPL 2. Currently, the capability is only used by fork processes to acquire the I/O database mutex.

The I/O database mutex basically synchronizes the lists of I/O data structures, for example, the linked list of UCBs associated with a particular device. A device driver that clones new device units from template devices must insert new units into the UCB list and remove units being deleted. Although these insertions and deletions are usually done from process context, in some cases they must be done from fork process context. For example, when the disk class driver fork process receives a message from a mass storage control protocol (MSCP) server that a new disk unit has come on line, it must clone a UCB and add it to the list.

The following routines in module MUTEX serve this need:

- SCH$LOCKWEXEC—Acquire write ownership of a mutex from a system thread
- SCH$LOCKREXEC—Acquire read ownership of a mutex from a system thread
- SCH$UNLOCKEXEC—Release a mutex from a system thread

The main difference between SCH$LOCKWEXEC and SCH$LOCKREXEC and their process context counterparts is that they return a failure status if the mutex is unavailable. There is no mechanism that transparently stalls a fork process and awakens it when the mutex becomes available. If a fork process receives a failure status, it must wait itself by using the fork and wait mechanism described in Chapter *Software Interrupts.*

These routines acquire the SCHED spinlock, which is held at IPL$_SCHED. This mechanism is therefore restricted to threads of execution that run at IPL 8 or below and that hold no higher ranking spinlock.

9.8 Further Information

- *VAX Architecture Reference Manual*, 2d ed., Chapter "Memory and I/O," for information on data sharing and synchronization implications
- *Computer Architecture: A Quantitative Approach,* by John L. Hennessy, David A. Patterson, and Morgan Kaufman, for background on cache protocols and cache and memory coherency

9.9 Revision Information

Source Module	Build Date
[SYS]FORKCNTRL	21-Mar-1991
[SYS]LDAT	21-Mar-1991
[SYS]MUTEX	21-Mar-1991
[SYS]SPINLOCKS	21-Mar-1991
[LIB]IPLDEF.SDL	21-Mar-1991
[LIB]MTXDEF.SDL	21-Mar-1991
[LIB]SPLDEF.SDL	21-Mar-1991
[LIB]SPLCODDEF.SDL	21-Mar-1991
[LIB]CPUDEF.SDL	21-Mar-1991
[LIB]SYSMAR.MAR	26-Jul-1991

Part VI/Input/Output

Chapter 22

Overview of the I/O Subsystem

Out of intense complexities intense simplicities emerge.

Winston Churchill

The major components of the VMS I/O subsystem are device drivers and their data structures, ancillary control processes, I/O support routines, and I/O system services. This chapter provides an overview of the VMS I/O subsystem.

The following chapters describe different aspects of the VMS I/O subsystem in more detail:

- Chapter *I/O Architecture* provides an overview of the [*censored*] I/O hardware architecture and the terms that describe its components.
- Chapter *I/O System Services* describes I/O system services and their use.
- Chapter *I/O Processing* describes driver I/O processing.
- Chapter *Mailboxes* describes software I/O mailboxes.
- Chapter *Miscellaneous I/O Topics* describes the terminal driver, pseudo devices, disk bad block processing, and other I/O-related topics.

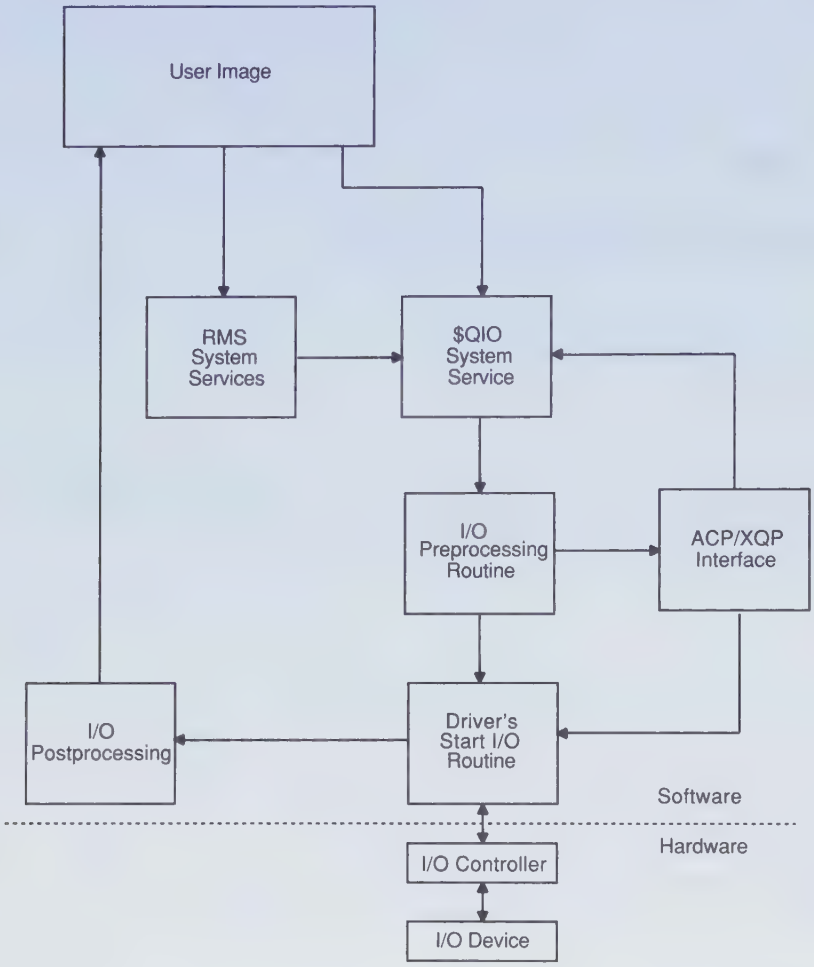
22.1 Overview

User images initiate most VMS I/O operations. The VMS executive initiates I/O operations for swapping, paging, file system, and other miscellaneous functions.

All I/O operation requests are eventually handled by a system software component called a device driver, or simply, a driver. The driver is primarily responsible for communicating system and user I/O requests to the I/O hardware and ensuring that they are carried out correctly within a reasonable time. Each type of device has a separate driver. Section 22.3 provides an overview of drivers.

Figure 22–1 shows various software components involved in user-initiated I/O operations. User images and most VMS components request an I/O operation through the Queue I/O Request ($QIO) system service. User images can indirectly request the $QIO system service using Record Management Services (RMS) services or a higher level language interface such as a FORTRAN `WRITE` statement or the C language `printf( )` function.

Figure 22–1 Overview of VMS I/O



The $QIO system service provides a device-independent user interface to drivers. If the device-independent parameters of a user's I/O request are valid, the service allocates and builds an I/O request packet (IRP). The IRP describes the I/O request and its context until the request completes successfully. The $QIO system service passes the IRP to the driver's I/O preprocessing routine for device- and function-specific validation. This I/O preprocessing routine may, for example, check the accessibility of the $QIO requestor's buffer in preparation for a direct memory access (DMA) transfer. $QIO and related system services are described in Chapter *I/O System Services*.

The driver's I/O preprocessing routine, also known as a function decision table (FDT) routine, delivers the IRP to the driver's start I/O routine if all the following conditions are met:

- The I/O request is valid.
- No ancillary control process (ACP) assistance is required.
- The requested function cannot be completed by the I/O preprocessing routine.

- No further I/O preprocessing is required.

Chapter *I/O System Services* elaborates on the steps taken by the driver's I/O preprocessing routine.

The start I/O routine interacts with the controller to perform the requested I/O operation. It and other driver routines are summarized in Section 22.3.2.

If the I/O preprocessing routine requires ACP assistance, it delivers the IRP to the ACP. An ACP assists a device driver with its complex functions. For example, the network ACP (NETACP) assists the network device driver (NETDRIVER), with creating links to other systems, performing routing and switching functions, and so on. The ACP can issue $QIO requests of its own to process a user's I/O request, or it can deliver the IRP to the driver's start I/O routine.

VMS provides I/O support routines that assist drivers at many levels. Section 22.5 summarizes these routines.

22.2 I/O Database

The I/O database serves the following main functions:

- It describes individual I/O hardware components, such as devices, controllers, adapters, and widgets, as well as device drivers.
- It describes system topology, that is, the configuration of and interrelations among I/O hardware components.
- It maintains the context of I/O operations.

This section gives a brief overview of the I/O database and the structures within it. Figure 22–2 shows the relations among the principal data structures in the I/O database.

22.2.1 I/O Hardware Components and Their Data Structures

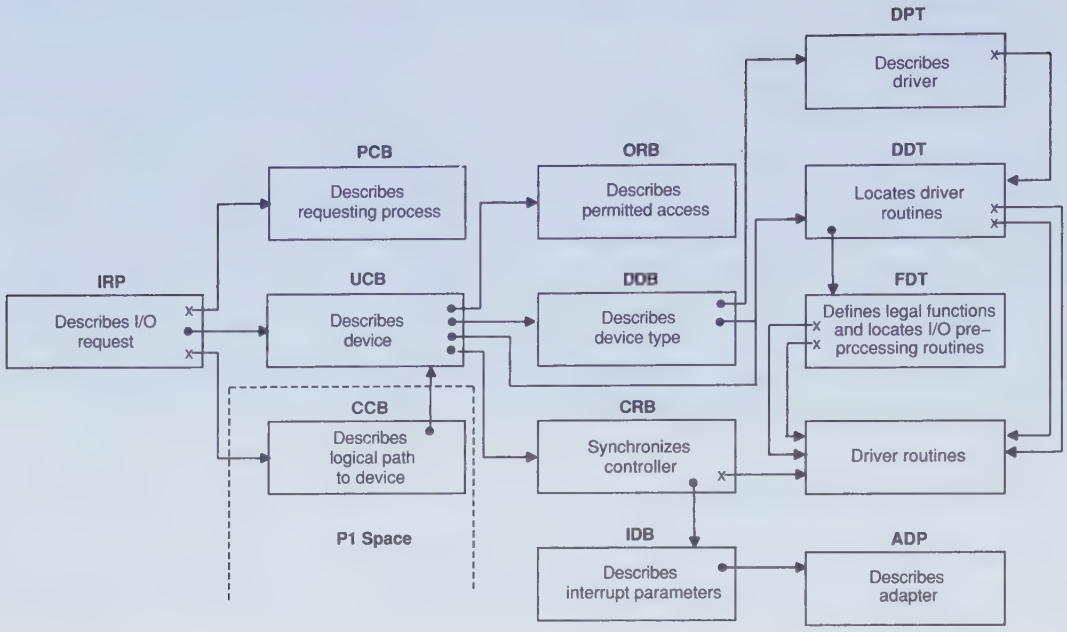
Each I/O hardware component has one or more data structures associated with it.

22.2.1.1 Data Structures for I/O Devices

The I/O database is unit-oriented. The item of interest to a process that requests the I/O operation is the target device unit. In most cases, the I/O controller, I/O adapter, and so on, are significant to the requesting process only because they communicate between the CPU and the device unit.

VMS creates a unit control block (UCB) for each I/O device attached to the system. A typical UCB, shown in Figure 22–3, defines the characteristics and current state of an individual device and is the focal point for controlling access to it. It contains a fork block (FKB) substructure (described in Chapter *Software Interrupts*). When a driver is stalled or interrupted, the driver's context is stored in the FKB. The UCB also contains pointers to various related data structures,

Figure 22–2 The I/O Database



and a queue header for the device's pending I/O requests. A UCB can have one or more device-specific extension areas.

VMS creates an object rights block (ORB) for each device when it creates the associated UCB. An ORB, pictured in Appendix *Data Structure Definitions*, describes the rights a process must have to access the object associated with the ORB. UCBs are not the only entities in VMS that have an associated ORB; although ORBs are a part of the I/O database, they are not unique to it.

A device data block (DDB) contains information common to all devices of the same type connected to a particular controller. A DDB is not used directly for controlling access to either the device controller or associated devices. Shown in Figure 22–4, a DDB records the generic device name (TT, for example) concatenated with the controller designator (A or B, for example). When combined with the unit number for the device, the generic device name forms the VMS name for the device (TTA10:, for example). A DDB also points to the driver prologue and driver dispatch tables.

IOC$GL_DEVLIST is the listhead for the list of all DDBs. Offset DDB$PS_LINK in each DDB points to the next DDB in the list. The last DDB in the list has zero in its DDB$PS_LINK field. Figure 22–5 shows the DDB list. Offset DDB$PS_UCB in each DDB points to the first in a list of UCBs for units of the same type attached to the controller associated with the DDB. The next UCB in this list is located through the UCB offset UCB$PS_LINK, as shown in Figure 22–5. The last UCB in the list has zero in its UCB$PS_LINK field.

Figure 22–3 Unit Control Block

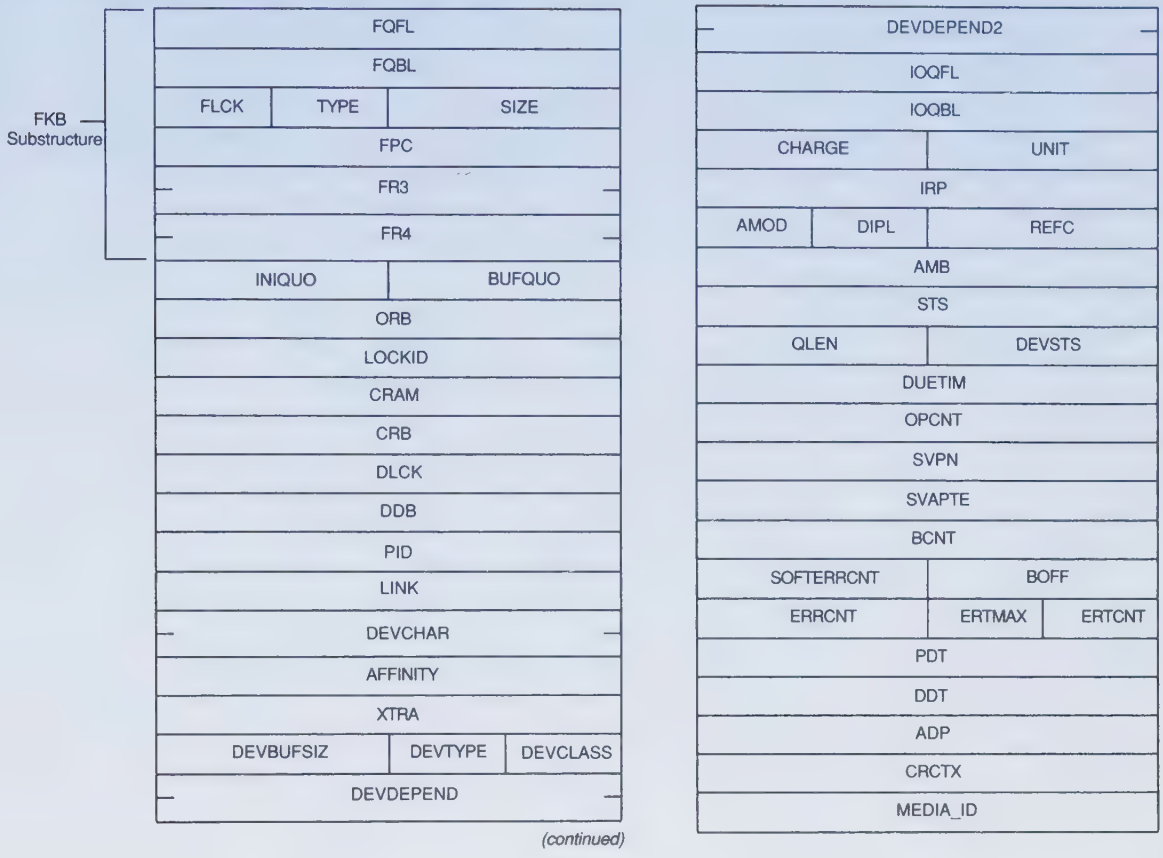
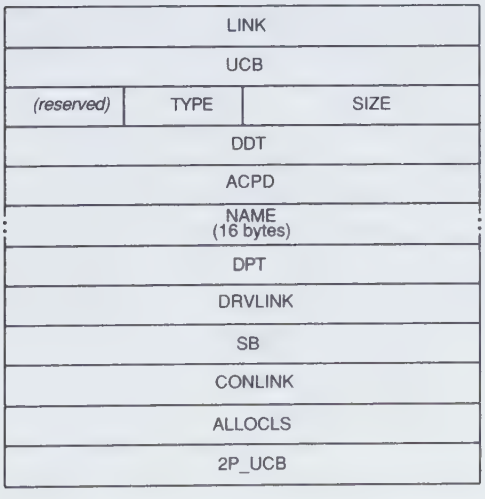


Figure 22–4 Device Data Block



The Assign Channel ($ASSIGN) system service, for example, searches the DDB list to locate the DDB corresponding to a user-specified generic device name. It then searches the DDB's UCB list to locate the UCB corresponding to the user-specified unit number.

All DDBs associated with a driver are linked together in a list that begins at offset DPT$PS_DDB_LIST in the driver prologue table (DPT). Offset DDB$PS_DRVLINK in each DDB points to the next DDB in this list, as shown in Figure 22–6. This list can be used to find all devices associated with a given driver. See Section 22.3.1.1 for more information on the driver prologue table.

Figure 22–5 DDB List

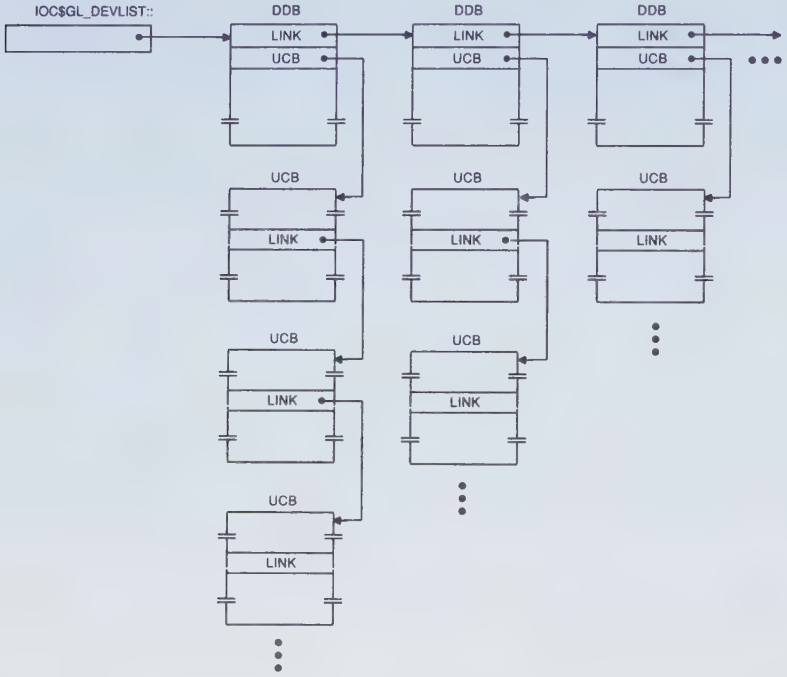
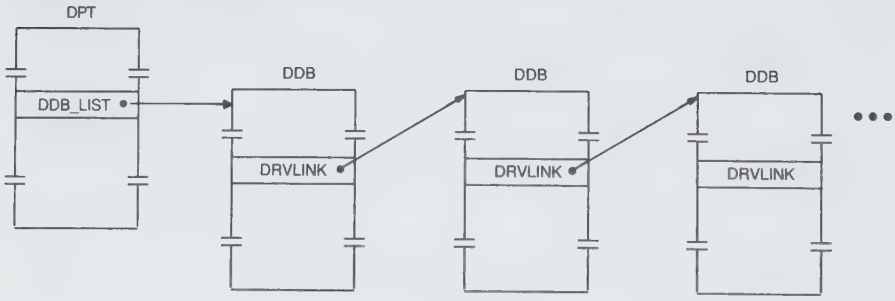


Figure 22–6 Driver's DDB List



22.2.1.2 Data Structures for I/O Controllers

VMS creates a channel request block (CRB) to represent and control access to an I/O controller. (A less ambiguous name for this data structure would be controller request block.) Shown in Figure 22–7, a CRB defines the current state of the controller. It contains an FKB substructure and a listhead of fork processes waiting for controller access. It also contains one interrupt transfer vector (VEC) substructure, shown in Figure 22–8, for each of the controller's interrupt vectors.

Figure 22–7 Channel Request Block

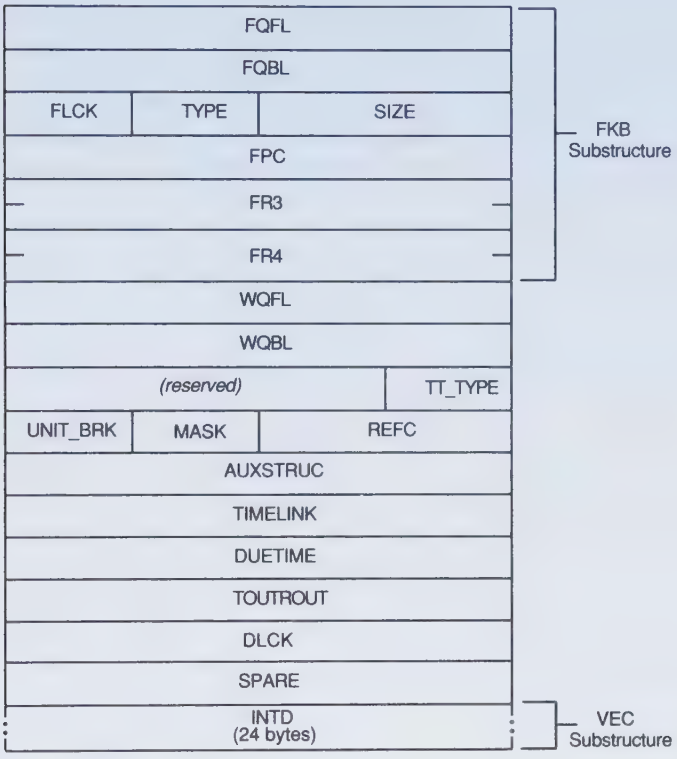
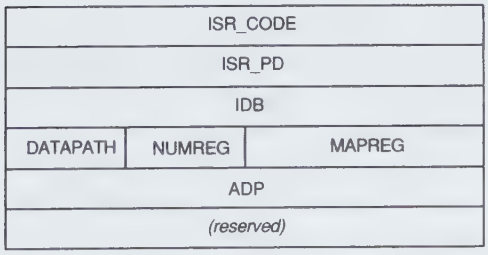


Figure 22–8 Interrupt Transfer Vector



A VEC substructure provides information about a device's interrupt: the location of its service routine and the address of the interrupt dispatch block (IDB) to be used by the service routine. The purpose and use of the VEC substructure are described in Chapter *I/O Processing*.

VMS creates an IDB for every CRB it creates. An IDB, shown in Figure 22–9, stores parameters for the device's interrupt service routine that enable it to determine which of the controller's units should be serviced. An IDB holds the address of the I/O controller's base control/status register (CSR), a list of controller register access mailboxes (CRAMs, described later in this section), and for some controllers, a list of UCBs associated with the controller.

An IDB also contains the system control block (SCB) vector offset for the controller's interrupts. If more than one interrupt vector is associated with the controller, its IDB points to an auxiliary structure called the vector list extension (VLE), shown in Figure 22–10. A VLE has an array of SCB vector offsets for the controller's interrupts. For more information, see Chapters *Hardware Interrupts and Machine Checks* and *I/O Processing*.

Figure 22–9 Interrupt Dispatch Block

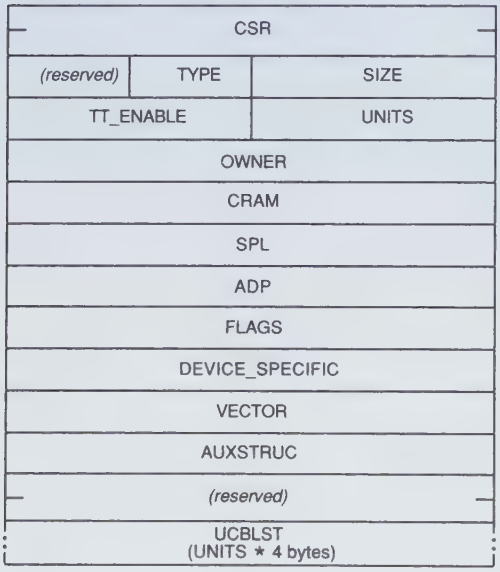
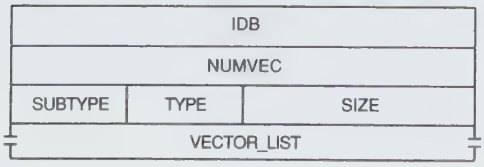
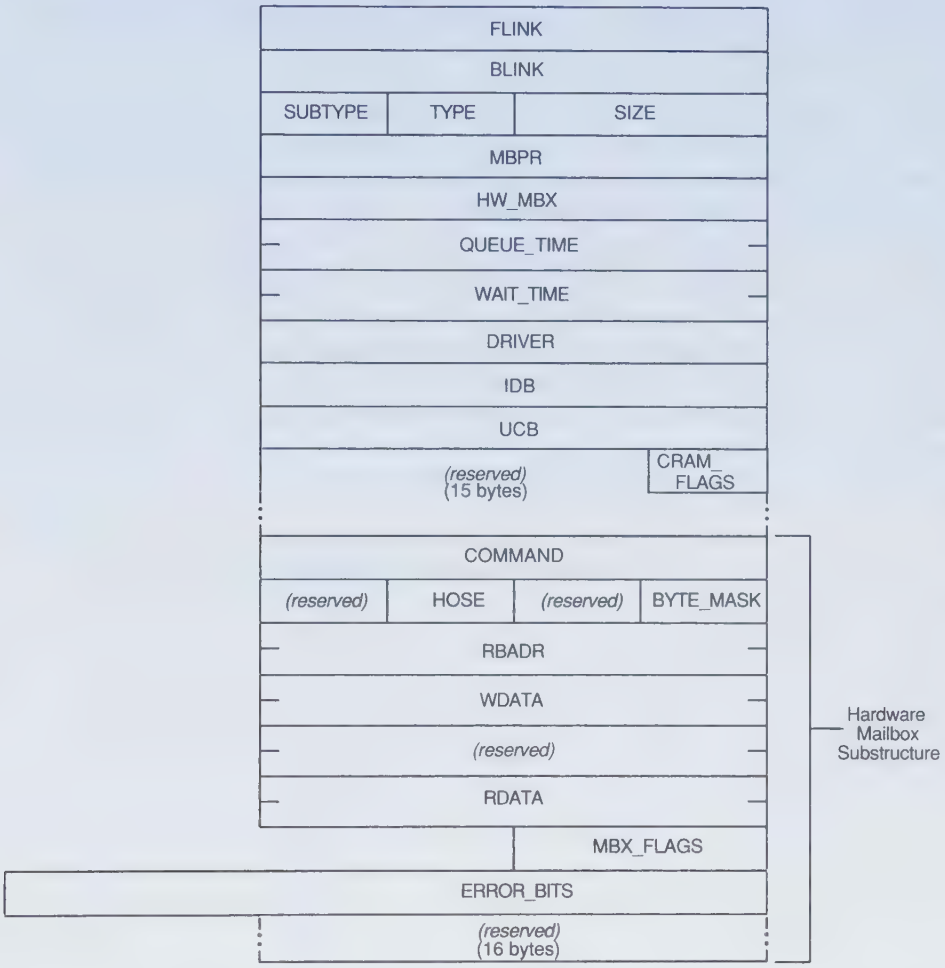


Figure 22–10 Vector List Extension



A CRAM can be associated with an I/O controller or an I/O device. Shown in Figure 22–11, a CRAM contains a hardware mailbox, a pointer to the local I/O adapter's mailbox pointer register (MBPR), and various parameters that govern the hardware mailbox access operation. Chapter *I/O Processing* describes hardware mailbox access support.

Figure 22–11 Controller Register Access Mailbox



22.2.1.3 Data Structures for Interconnects, I/O Adapters, and I/O Widgets

In VAX/VMS an adapter control block (ADP) simply represents a hardware block that connects one interconnect to another. In [*censored*] VMS, the ADP's function has been extended to include information about the interconnect and the system configuration.

Despite its name, therefore, an ADP, shown in Figure 22–12, can now represent one of the following: the processor-memory interconnect (PMI), a tightly coupled I/O interconnect, or a multichannel I/O widget.

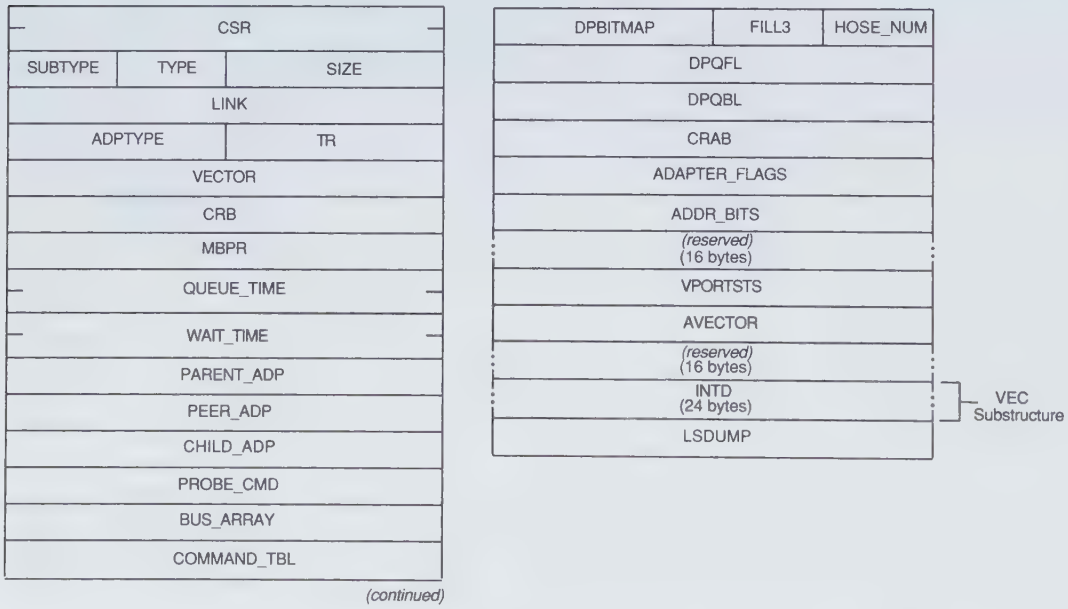
The system ADP represents the PMI. It has an auxiliary bus array structure that describes all PMI nodes. An ADP other than the system ADP represents either a tightly coupled I/O interconnect or a multichannel I/O widget.

An ADP for a tightly coupled I/O interconnect contains information related to hardware mailbox support, system topology, adapter interrupts, and related items. It also contains information about the I/O adapter that connects the interconnect to the PMI or to a parent tightly coupled I/O interconnect. The adjective *parent* in this context describes the tightly coupled I/O interconnect that is closer to the PMI.

Information relating to an I/O widget is normally only maintained in a widget-specific data structure defined and used by the widget's driver. Information that is common to all loosely coupled I/O interconnects connected to a multichannel I/O widget is, however, maintained in an ADP.

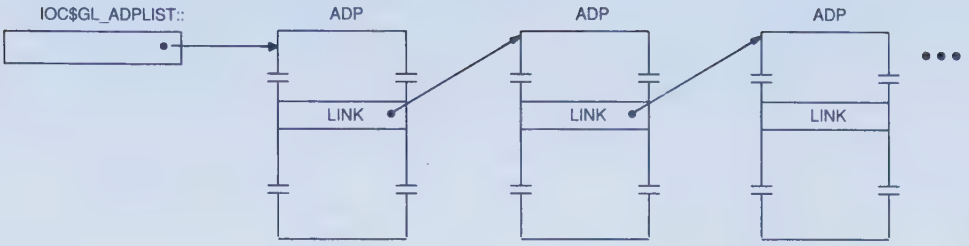
An ADP can have up to four auxiliary data structures: adapter bus array (BUSARRAY), pointed to by ADP$PS_BUS_ARRAY; adapter command table (CMDTABLE), pointed to by ADP$PS_COMMAND_TBL; counted resource allocation block (CRAB), pointed to by ADP$PS_CRAB; and interrupt vector table, pointed to by ADP$PS_VECTOR. The BUSARRAY structure is described later in this section. For more information on the other three structures, see Chapter *I/O Processing*.

Figure 22–12 Adapter Control Block



IOC$GL_ADPLIST is the listhead for the list of all ADPs, shown in Figure 22–13. The first ADP on this list is always the system ADP. Offset ADP$PS_LINK in each ADP points to the next ADP in the list. The last ADP in the list has zero in its ADP$PS_LINK field. In response to the command SHOW ADAPTER, the IOGEN utility traverses and displays this list.

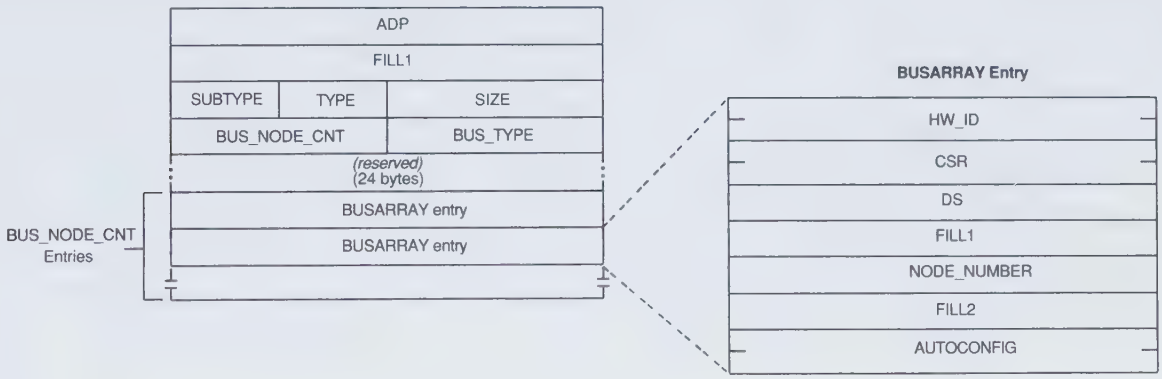
Figure 22–13 ADP List



The hierarchy of tightly coupled I/O interconnects on a system is represented by the interconnections between ADPs in the ADP list. In conjunction with the auxiliary BUSARRAY structure (shown in Figure 22–14) for each ADP, this information represents a system's configuration. For example, Figure 22–15 shows how ADPs and their auxiliary BUSARRAY structures represent the SYSTEM_A configuration.

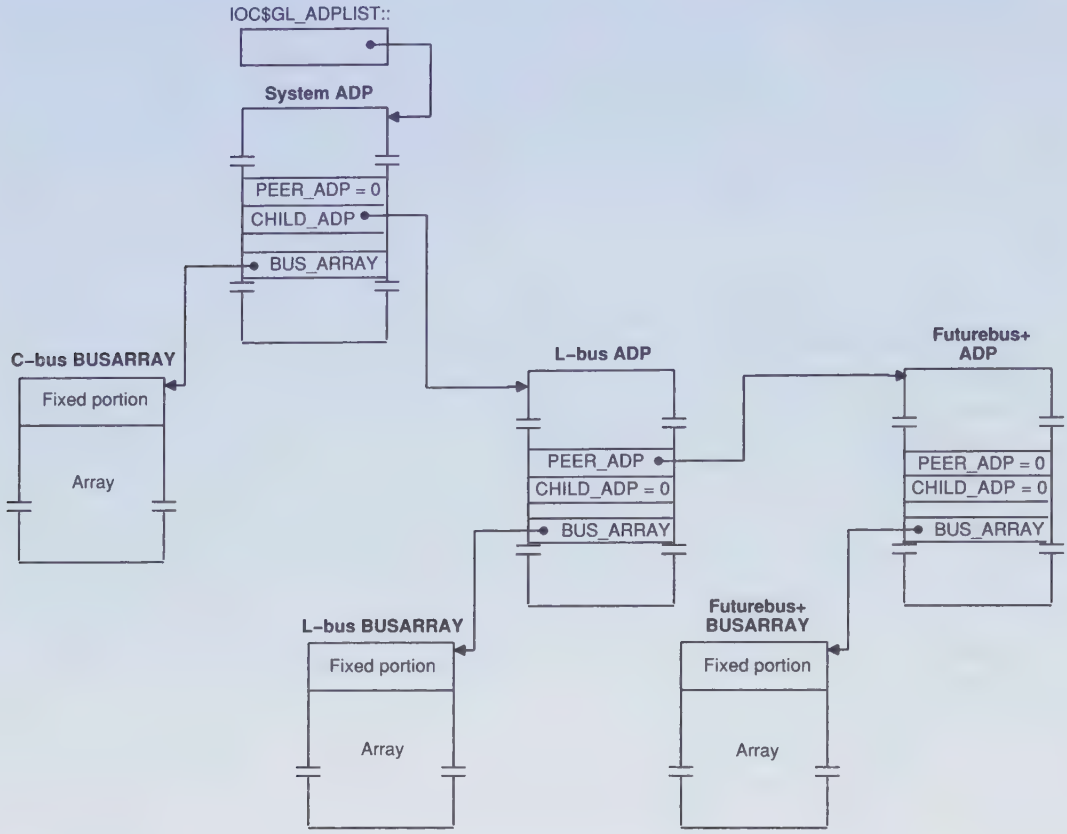
At the root of the hierarchical ADP list is the system ADP. Offset ADP$PS_CHILD_ADP in the system ADP points to an ADP for a tightly coupled I/O interconnect at the next level in the hierarchy—one that connects to the PMI directly, that is, without other intervening tightly coupled I/O interconnects. In Figure 22–15, the system ADP's ADP$PS_CHILD_ADP points to the local bus (L-bus) ADP.

Figure 22–14 ADP Bus Array



Offset ADP$PS_PEER_ADP in the system ADP always contains the value zero because the PMI has no peers. The SYSTEM_A system L-bus and Futurebus+ are both tightly coupled I/O interconnects that are directly connected to the C-bus through the SYSTEM_A I/O module. Offset ADP$PS_PEER_ADP in the L-bus ADP points to the Futurebus+ ADP, because the Futurebus+ is the L-bus's peer, that is, it connects to the system at the same level as the L-bus. ADP$PS_CHILD_ADP in each of the L-bus and Futurebus+ ADPs contains the value zero.

Figure 22–15 ADP Hierarchy and System Configuration

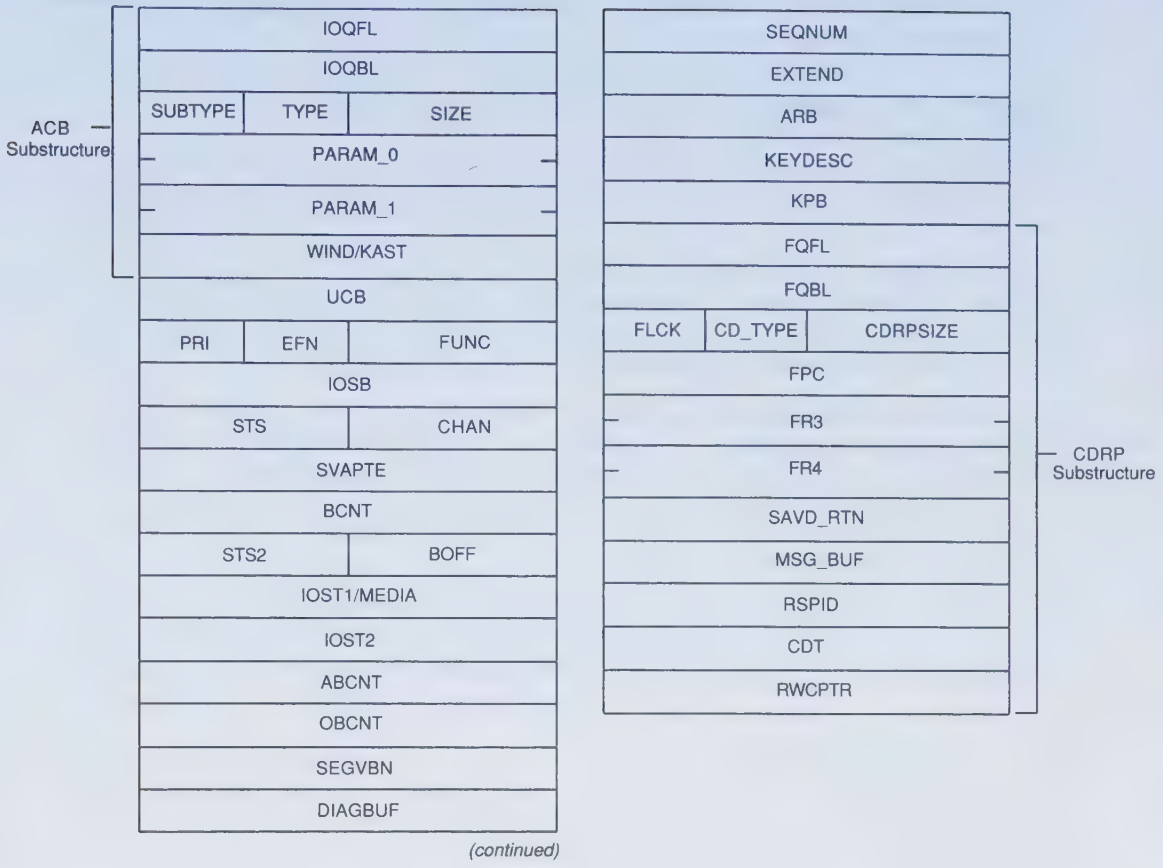


The BUSARRAY structure contains information about nodes on a tightly coupled I/O interconnect. It consists of a fixed portion and an array of entries. The fixed portion contains the tightly coupled I/O interconnect type, the number of nodes, and a pointer to the ADP. There is one array entry for each tightly coupled I/O interconnect node. Each entry records the node number, hardware identification of the node, and a pointer to either an ADP or a CRB. If the pointer points to an ADP, then the node is either an I/O adapter or a multichannel I/O widget. If the pointer points to a CRB, the node is an I/O widget.

22.2.2 Context of I/O Operations

VMS maintains the context of user I/O requests in the channel control block (CCB) and the IRP, the latter pictured in Figure 22–16. Before an image can perform I/O operations to a device, it must assign an I/O channel to the device. It does this by requesting the $ASSIGN system service, which allocates a CCB to describe the channel. Unlike other I/O data structures described here, CCBs are located in the P1 space of each process. An I/O channel, described by the CCB, is the software mechanism that links a process to the target device of an I/O operation. The I/O channel and CCB are described in Chapter *I/O System Services*.

Figure 22–16 I/O Request Packet



When an image requests the $QIO system service for an I/O operation, the $QIO system service procedure, in cooperation with the driver's preprocessing routine, constructs an IRP to describe the I/O request in a standard format. The device-independent arguments of the I/O request are recorded in the IRP. Additionally, if the operation involves data transfer, the user buffer's address and size are recorded in the IRP. The IRP has an asynchronous system trap (AST) control block (ACB) substructure, a pointer to the UCB for the target device, addresses of other related data structures, and a class driver request packet (CDRP) substructure. See Section 22.3 for information on class drivers.

The CDRP substructure describes a request to be handled by a system communication services (SCS) port driver. Such requests are generated, for example, by the mass storage control protocol (MSCP) disk and tape class drivers. To save time in these drivers, each IRP includes enough space for a CDRP, although the space is not always used for that purpose.

When the I/O subsystem initiates an I/O operation, it creates a fork process thread of execution in which driver code executes. The fork process's context is maintained in a fork block, which is usually a substructure of a larger structure, for example, a UCB or a CDRP. A driver fork process that requires more context to be maintained can execute as a kernel process, a thread of execution with

a private stack. A kernel process block (KPB) describes each kernel process. Chapter *Software Interrupts* contains further details on fork and kernel processes, FKBs, and KPBs.

22.2.3 Synchronizing Access to the I/O Database

The following methods are used to synchronize access to the I/O database: mutexes, interrupt priority level (IPL), spinlocks, and the lock management system services.

The I/O database mutex, IOC$GL_MUTEX, synchronizes access to the I/O database. This mutex does not synchronize access to any of the hardware components of the I/O subsystem. Its major purpose is to synchronize the addition or deletion of data structures with searches of the I/O database.

The spinlocks of most interest to the I/O subsystem are fork locks and device locks. Fork locks synchronize fork processing. A device lock synchronizes access to the device controller data structures and thus to the controller.

IPL synchronization of the I/O database normally occurs as part of spinlock acquisition and release.

Chapter *Synchronization Techniques* discusses the use of IPL, spinlocks, and mutexes for synchronization. Chapter *Lock Management* describes lock management system services.

22.3 Device Drivers

A device driver is a collection of routines and tables that assist VMS in performing I/O operations on a device. The following are its key functions:

- Defining an I/O device for the rest of the operating system
- Preparing an I/O device and its controller for operation during system initialization, during connection of the device by the driver-loading procedure, and during recovery from a power failure
- Validating device-dependent parameters of a user's I/O request
- Translating user I/O requests into device-specific commands
- Activating the I/O device by accessing one or more interface registers
- Responding to device interrupts
- Responding to device timeout conditions
- Responding to I/O request cancellations
- Logging errors
- Returning device status to the I/O requestor

There are two driver models in VMS—the traditional model, in which the entire driver is contained in one image, and the class/port model, in which the driver consists of more than one image.

In the class/port model, the class driver performs functions common to a class of device, such as an MSCP disk or a small computer systems interconnect (SCSI) tape, for example. The port driver contains controller-specific subroutines for the class driver. For any given device, a class driver is bound to a port driver through the UCB or a port-specific data structure. The functional division into class and port drivers simplifies maintenance and integration of software when new devices and controllers are introduced.

The following are different categories of class/port driver, distinguished on the basis of the interaction between class and port driver:

- MSCP drivers
- SCSI drivers
- Terminal drivers
- Communications drivers

Class/port drivers are discussed in more detail in Chapter *Miscellaneous I/O Topics*. The term *driver* when used by itself usually refers to a traditional VMS driver.

22.3.1 Driver Tables

The following tables—driver prologue table, driver dispatch table, and function decision table—are included in most drivers.

22.3.1.1 Driver Prologue Table

A driver prologue table, shown in Figure 22–17, is a statically defined data structure within the driver image. It contains information that helps SYS$LOAD_DRIVER, the driver-loading procedure, in loading the driver into memory and creating the driver's I/O database. For example, a DPT defines the size of the device's UCB, the number of per-unit and per-controller CRAMs, and the driver's name. It also points to the driver dispatch table and to procedure descriptors for various driver routines that assist the driver-loading procedure in preparing the device's I/O database.

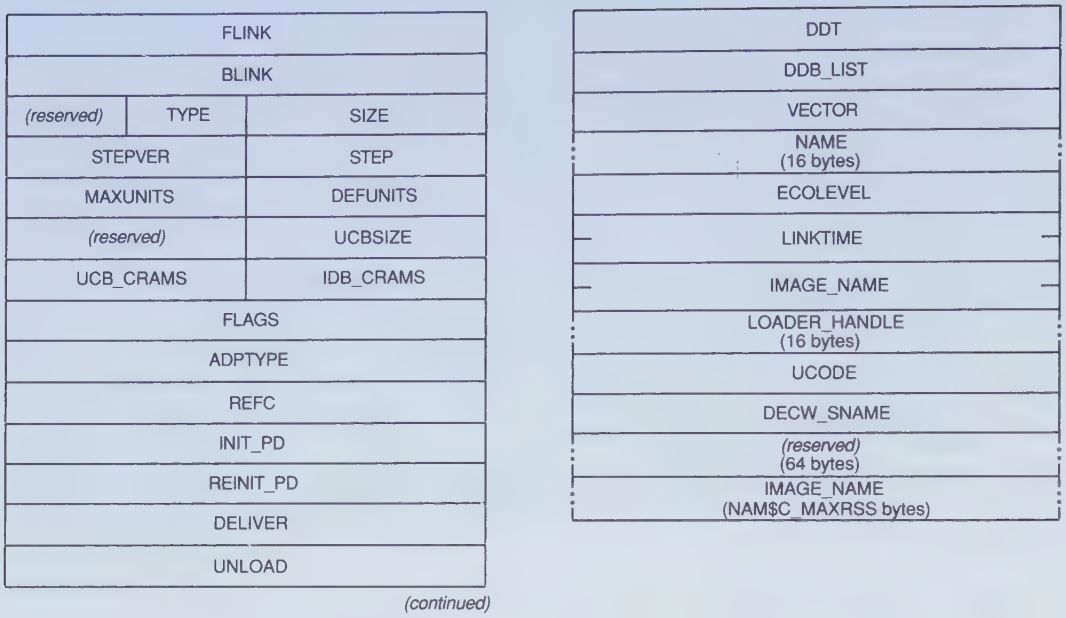
After loading the driver, SYS$LOAD_DRIVER chains the DPT to the systemwide list of DPTs starting at global location IOC$GL_DPTLIST.

22.3.1.2 Driver Dispatch Table

Each device driver has a set of standard routines, described in Section 22.3.2, that the VMS executive can invoke to perform device-related operations. VMS locates these routines through the driver dispatch table (DDT), shown in Figure 22–18.

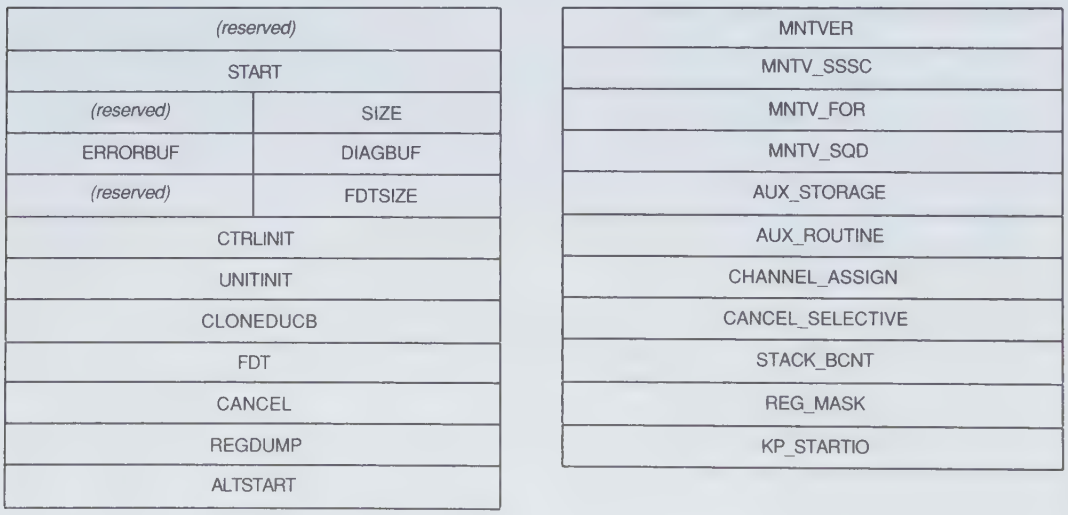
A DDT is a statically defined data structure within the driver image. In addition to containing procedure values for a driver's start I/O routine, controller initialization routine, unit initialization routine, and other routines, a DDT also points to the driver's function decision table and defines the sizes of a driver's diagnostic and error log buffers.

Figure 22–17 Driver Prologue Table



(continued)

Figure 22–18 Driver Dispatch Table



(continued)

22.3.1.3 Function Decision Table

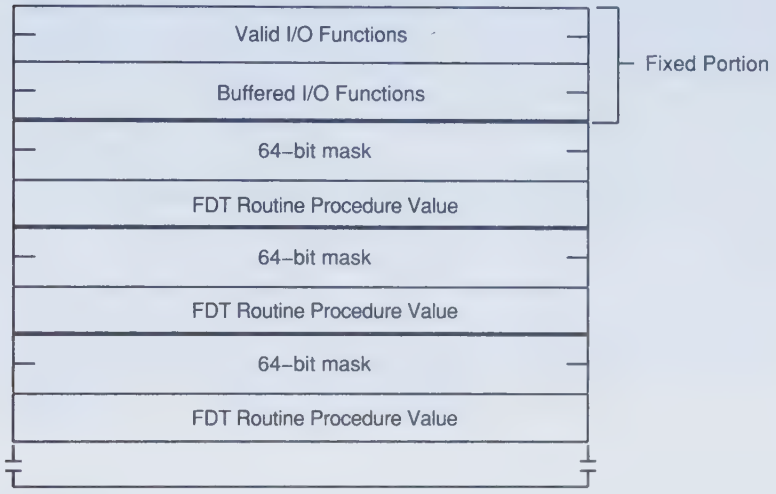
A function decision table, shown in Figure 22–19, lists all valid I/O function codes for the device and associates valid codes with the addresses of I/O preprocessing routines, also known as FDT routines.

Statically defined within the driver image, an FDT consists of a fixed portion followed by an array of FDT entries. The fixed portion consists of two 64-bit masks: the legal function mask, and the buffered function mask.

I/O function codes have values ranging from 0 to 63. Each bit in a mask corresponds to an I/O function code. For example, bit 33 in a mask corresponds to I/O function code 33. Bits set in the legal function mask indicate which function codes are supported for the associated devices. Bits set in the buffered function mask indicate which function codes represent buffered I/O operations.

Each FDT entry consists of a 64-bit mask and the procedure value of an FDT routine. The $QIO system service invokes the FDT routine if the bit corresponding to a requested function is set in the associated mask. An FDT routine validates the device- and function-dependent parameters of an I/O request and prepares the IRP for the driver's start I/O routine (see Section 22.5).

Figure 22–19 Function Decision Table



22.3.2 Driver Routines

Table 22–1 provides a summary of VMS driver routines. It classifies driver routines loosely into the following categories based on their major functions:

- Routines that prepare the I/O database

  With the exception of the cloned UCB routine, these routines are invoked by SYS$LOAD_DRIVER to set up and maintain the driver's I/O database.

  After loading a driver, SYS$LOAD_DRIVER creates the following data structures for the device and its associated components, based on information in the DPT: DDB, UCB, ORB, CRB, IDB, VLE, and possibly CRAMs. If the CRB and IDB were already created during system initialization, they are not recreated.

  SYS$LOAD_DRIVER initializes device-independent fields in the device's I/O database, that is, the device's associated I/O data structures. It then calls the driver's I/O database initialization routine, which initializes various device-specific fields in the I/O database. SYS$LOAD_DRIVER also calls the

driver's I/O database reinitialization routine, which initializes fields in the I/O database that could change if the driver were reloaded.

Traditionally, VAX/VMS initialization of a device's I/O database has been divided into fields initialized only once when the device driver is first loaded and those initialized again whenever the driver is reloaded. Although [*censored*] VMS does not currently support driver reloading, it reflects VAX/VMS in providing both kinds of initialization.

If a driver provides a unit delivery routine, the VMS autoconfiguration procedure calls it to determine whether to configure a given device unit.

EXE$ASSIGN, the $ASSIGN system service procedure, invokes a device's cloned UCB routine when an image assigns an I/O channel to a software template device such as the NET: device (see Chapter *I/O System Services*). The cloned UCB routine performs device-specific initialization of the software device's cloned UCB.

- Routines that prepare controllers and devices

  A controller initialization routine prepares a controller for I/O operations by initializing its interface registers, permanently allocating controller-specific resources, and performing other necessary steps. It is invoked by SYS$LOAD_DRIVER.

  A unit initialization routine prepares an individual device unit for operation; it is invoked by SYS$LOAD_DRIVER for each unit of a multiunit controller. It also initializes device-specific fields of the UCB and sets the UCB$V_ONLINE bit to make the device usable by software.

- Routines that prepare user I/O requests

  An image requests the $QIO system service for each I/O operation. EXE$QIO, the $QIO system service procedure, validates the request using the legal and buffered function masks in the FDT (see Section 22.3.1.3). Subsequently, it allocates and builds the device-independent portion of the IRP. The function-specific parameters of the I/O request can only be processed and entered into the IRP with the help of FDT routines, function-specific routines in the driver.

Table 22–1 Summary of Driver Routines

Routine	Function	Located Through	Called/Invoked by
	Routines That Prepare the I/O Database		
I/O database initialization	Prepares the driver's I/O database when a driver is loaded	DPT$PS_INIT_PD	SYS$LOAD_DRIVER
I/O database reinitialization	Initializes the part of a driver's I/O database that would be reinitialized if the driver were reloaded	DPT$PS_REINIT_PD	SYS$LOAD_DRIVER

Table 22–1 (Cont.) Summary of Driver Routines

Routine	Function	Located Through	Called/Invoked by
Routines That Prepare the I/O Database			
Unit delivery	For a multiunit controller, determines which units exist	DPT$PS_DELIVER	SYS$LOAD_DRIVER
Cloned UCB	Performs device-specific initialization of a pseudo device's cloned UCB	DDT$PS_CLONEDUCB	EXE$ASSIGN
Routines That Prepare Controllers and Devices			
Controller initialization	Prepares a controller for operation	DDT$PS_CTRLINIT	IOC$CTRLINIT
Unit initialization	Prepares a device for operation; for a single unit controller, prepares the controller as well	DDT$PS_UNITINIT	IOC$UNITINIT
Routines That Prepare User I/O Requests			
I/O preprocessing (FDT)	Performs function- and device-specific I/O request preprocessing	FDT entries	EXE$QIO
Routines That Process I/O Requests			
Start I/O	Initiates and processes an I/O operation on a device	DDT$PS_START	IOC$INITIATE, IOC$REQCOM
Alternate start I/O	Initiates and processes an I/O request on a device whose driver supports multiple concurrent I/O operations	DDT$PS_ALTSTART	EXE$ALTQUEPKT
Interrupt service	Services device interrupts	VEC substructure in CRB or ADP	IO_INTERRUPT
Cancel I/O	Cancels an in-progress I/O request	DDT$PS_CANCEL	EXE$CANCEL, EXE$DALLOC, EXE$DASSGN
Register dumping	Copies interface registers to a diagnostic or error log buffer	DDT$PS_REGDUMP	ERL$DEVICERR, ERL$DEVICTMO, ERL$DEVICEATTN, IOC$DIAGBUFILL

- Routines that process I/O requests

  A start I/O routine is the starting point for the system thread of execution that carries an I/O request through to completion. It is entered by the VMS executive after an I/O request has been validated. It translates a function request to a device-specific command and initiates device action, typically by writing to one or more device interface registers, sending command messages, or other means. Subsequently, it waits for the device to signal completion of device action through an interrupt, an end message, or other means. The device's interrupt service routine responds to a device interrupt by reactivating the thread of execution that was responsible for the device interrupt. When device processing completes successfully, or when an error or timeout condition occurs, a start I/O routine initiates I/O request post-processing to post the status of the I/O request to the requestor. Chapter *I/O Processing* details the actions of a driver's start I/O routine.

  An alternate start I/O routine performs actions similar to a start I/O routine except that it supports the processing of multiple concurrent I/O requests.

  A cancel I/O routine provides driver-specific processing to cancel an I/O request in progress, that is, one currently being processed by the start I/O routine. The Cancel I/O ($CANCEL) system service can cancel pending (that is, not yet in progress) I/O requests because no driver-specific processing is required; it simply deallocates IRPs from the pending I/O queue of the device UCB.

  Drivers log errors to the system error log file typically through one of three routines: ERL$DEVICERR, ERL$DEVICTMO, or ERL$DEVICEATTN, all in module ERRORLOG (see Chapter *Error Handling*). These routines, as well as IOC$DIAGBUFILL, in module IOSUBNPAG, invoke a driver's register dumping routine. IOC$DIAGBUFILL provides support to drivers that perform diagnostic I/O. The register dumping routine copies device interface registers and other important information to the error log or diagnostic buffer that its invoker allocates.

22.4 Ancillary Control Processes

A driver executes mostly at elevated IPL with minimal context; it cannot request system services. This makes it difficult to implement complex functions in drivers. An ACP is a separate thread of execution running in process context that implements complex driver functions.

As shown in Figure 22–1, a driver's FDT routine passes an I/O request either to an ACP or to a driver's start I/O routine, depending on the I/O function requested. A complex function request such as opening a disk file or establishing a network logical link, for example, is typically handled by an ACP.

An ACP image usually hibernates, waiting to be awakened to service I/O requests. An I/O request is passed to the ACP by queuing the IRP to the ACP queue block (AQB), shown in Figure 22–20, and waking up the ACP if necessary.

VMS provides the following ACPs:

- MTAACP—Magnetic tape ACP

- NETACP—DECnet–VAX ACP
- REMACP—Remote terminal ACP
- F11BXQP—Files-11 on-disk structure level 2 extended QIO processor

In VAX/VMS Version 4, the Files-11 on-disk structure level 2 ACP, F11BACP, was converted to F11BXQP, also simply called the XQP. The XQP implements the Files-11 on-disk structure level 2 on disks, and assists disk drivers with file system function requests. Unlike other ACPs, the XQP runs in the context of the process that issued the I/O request. The mechanism of IRP delivery to the XQP is also different, as briefly described in Chapter *I/O Processing*.

For the purposes of this chapter, the XQP is simply another form of ACP; unless explicitly mentioned, the term *ACP* also refers to the XQP.

Two data structures are common to most ACPs—an AQB, which holds information specific to the ACP, and a volume control block (VCB), which describes a volume that is serviced by an ACP.

An AQB has a queue header for IRPs waiting to be processed, the process ID of the ACP, the ACP's class and type, and a mount count, which typically indicates the number of volumes the ACP is servicing. All AQBs on a system are linked together. The global location IOC$GL_AQBLIST is the listhead for this list. Offset AQB$L_LINK points to the next AQB in the list, as shown in Figure 22–21.

Figure 22–20 ACP Queue Block

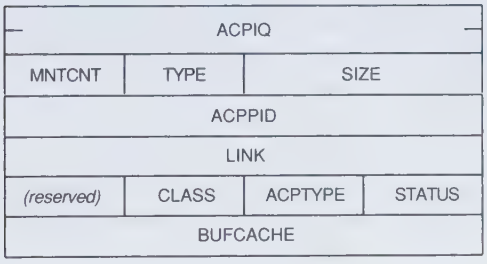
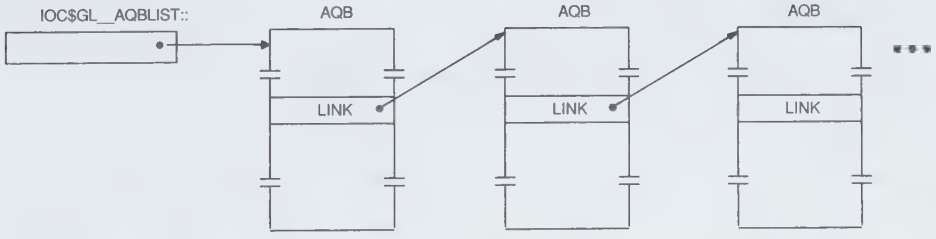


Figure 22–21 AQB List



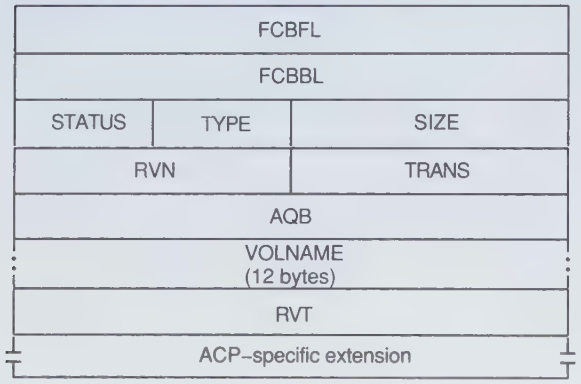
The term *volume* refers to a set of one or more devices that is treated as a single entity by an ACP; an ACP is responsible for maintaining the structure of data on a volume. The following are examples of a volume:

- A disk unit, serviced by the XQP
- A set of disk units (also called a volume set), serviced by the XQP
- The DECnet network, serviced by NETACP
- A tape unit, serviced by MTAAACP

A VCB describes a volume. A device's UCB points to the VCB for the volume to which the device belongs. A VCB may be known by a different name within the ACP. For example, the DECnet ACP's VCB is known as a routing control block (RCB). Typically, a VCB contains information about the organizational structure of a device.

A VCB has a fixed portion, shown in Figure 22–22, followed by an ACP-specific extension. One example of an ACP-specific extension is the VCB disk extension, shown in Figure 22–23, which is used by the XQP. Another example, shown in Figure 22–24, is the VCB magnetic tape extension, used by MTAAACP. Figure 22–25 shows how a device's VCB and AQB can be located.

Figure 22–22 Volume Control Block

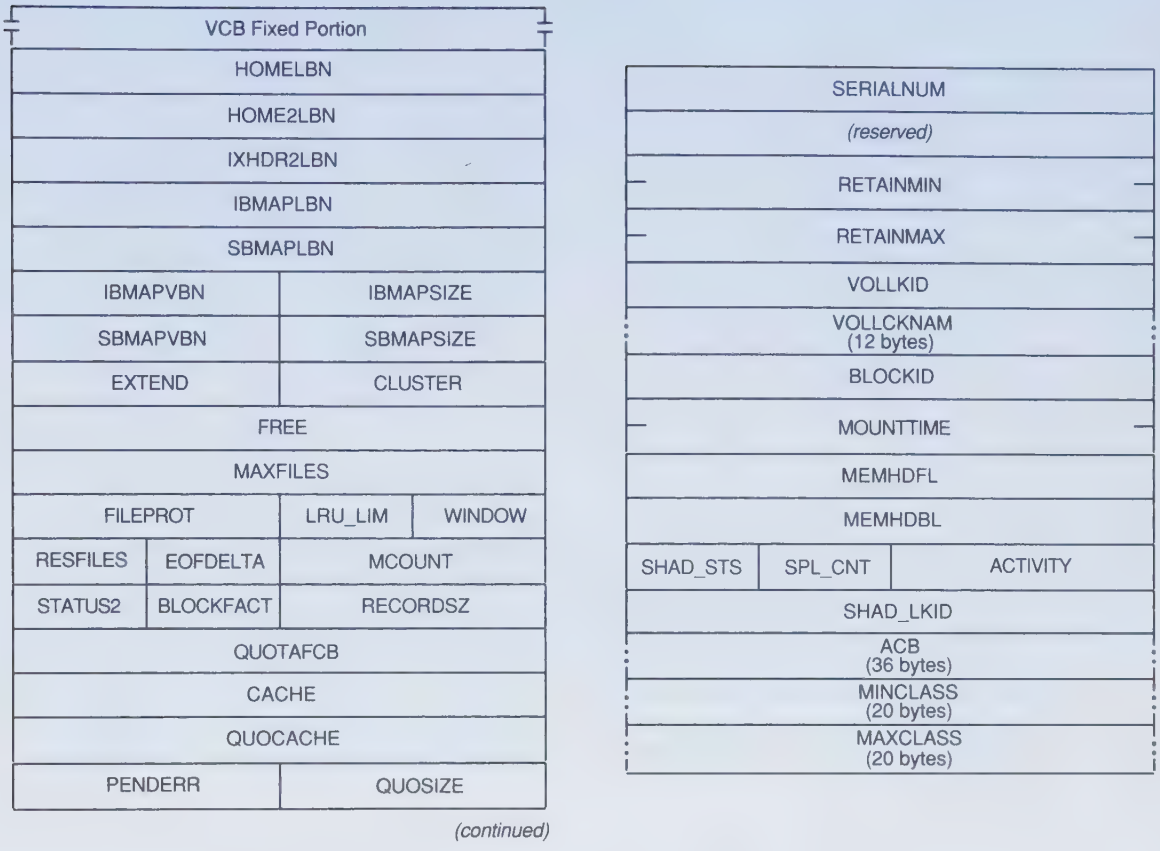


22.5 I/O Support Routines

VMS provides a number of I/O support routines that enable common driver functions to be performed in a consistent fashion. These routines perform a variety of functions needed by drivers. They are categorized as follows:

- I/O preprocessing support routines. These FDT routines perform preprocessing for common functions such as validating a user-specified buffer, locking a user buffer into memory, translating a logical or virtual function request to the corresponding physical function request, and copying function-specific parameters to the IRP.

Figure 22–23 VCB Disk Extension



(continued)

Figure 22–24 VCB Magnetic Tape Extension

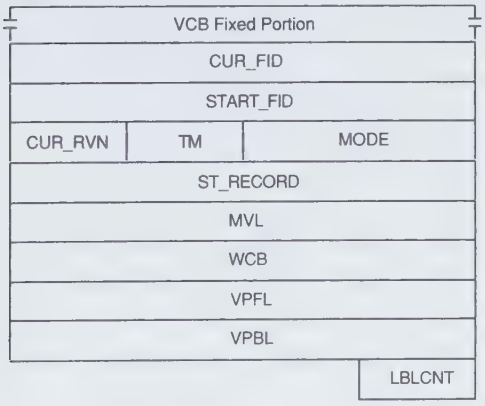
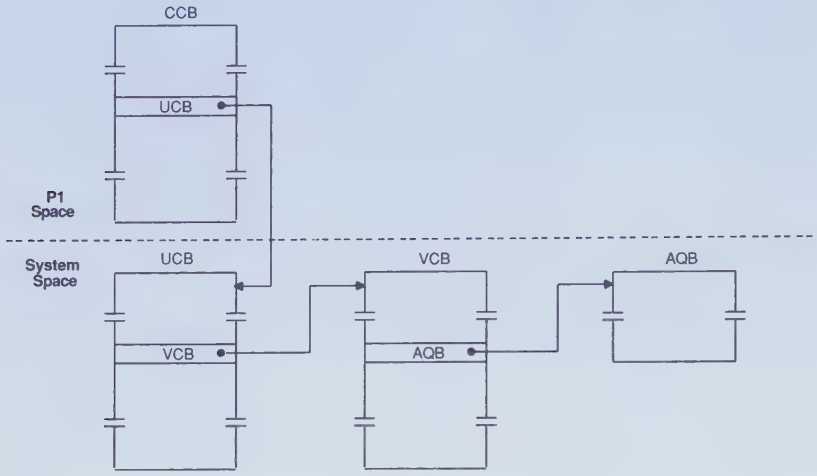


Figure 22–25 Locating a Device's VCB and AQB



- Fork and kernel process support routines. A driver's start I/O routine invokes these routines for fork or kernel process support. As described in Chapters *Software Interrupts* and *I/O Processing*, VMS creates a fork process for every I/O operation it initiates. The fork process can execute as a kernel process if it requires additional context.
- I/O request completion support routines. These routines complete driver I/O operations. Some are invoked by a driver's FDT routine; others by a driver's start I/O routine.
- Hardware mailbox support routines. These routines manage the allocation and use of hardware mailboxes. Architectural aspects of hardware mailbox access are outlined in Chapter *I/O Architecture*.
- Spinlock support routines. Drivers acquire and release static and dynamic spinlocks using these routines, which are described in Chapter *Synchronization Techniques*.
- Diagnostic and error logging support routines. These routines assist in debugging drivers and in reporting and recording device-specific errors. Chapter *Error Handling* describes error logging.
- Timer queue and timed wait support routines. These routines assist drivers in allocating, inserting, and removing timer queue entries, and implementing timers and interconnect-specific delay intervals. Chapter *Time Support* describes these routines.
- Buffer allocation support routines. These routines assist drivers in allocating nonpaged pool, and in enforcing user process quotas. Chapter *Pool Management* summarizes routines that allocate nonpaged pool.

A number of miscellaneous routines assist drivers with various other tasks as well.

Chapter 23

I/O System Services

Delay not Caesar! Read it instantly!

Shakespeare, *Julius Caesar,* 3, i

Here is a letter, read it at your leisure.

Shakespeare, *Merchant of Venice,* 5, i

An image performs I/O operations on a device by requesting I/O system services. System components such as Record Management Services (RMS), ancillary control processes (ACPs), and the Files-11 Extended QIO Processor (XQP) also request I/O system services on behalf of a process. This chapter describes the basic I/O system services and the device-independent portions of the flow of an I/O request. Chapter *I/O Processing* describes the device-dependent portion of that flow.

23.1 Overview

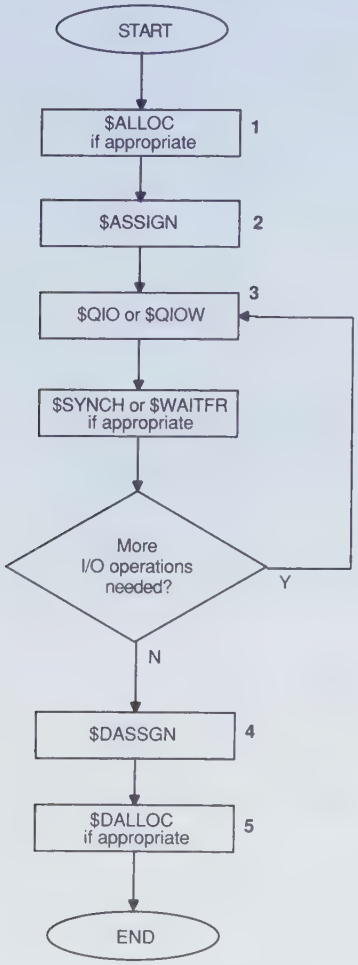
The basic I/O system services are

- Allocate Device ($ALLOC), by which an image reserves a particular device for exclusive use
- Deallocate Device ($DALLOC), by which an image relinquishes such a device
- Assign I/O Channel ($ASSIGN), by which an image creates a logical link to a device
- Deassign I/O Channel ($DASSGN), by which an image deletes the logical link
- Queue I/O Request [and Wait] ($QIO[W]), by which an image requests an I/O operation on a particular logical link to a device
- Cancel I/O on Channel ($CANCEL), by which an image cancels outstanding I/O requests on a particular logical link to a device

VMS provides other I/O system services in addition to those discussed in this chapter.

All the system service routines discussed in this chapter that have a device name argument also accept a logical name instead of a device name. Each routine uses the same criteria to process the device name argument. *Introduction to VMS System Services* discusses these criteria. Logical names and logical name translation are discussed in Chapter *Logical Names*.

Figure 23–1 Typical Use of I/O System Services



A typical service sequence for an image is shown in Figure 23–1; the numbers in the figure correspond to those in the following list:

❶ If appropriate, the image requests the $ALLOC system service. The appropriateness is determined as explained in Section 23.4.

❷ It requests the $ASSIGN system service.

❸ Either it requests the $QIO system service followed by an event flag wait system service (for example, Wait for Single Event Flag, $WAITFR, or Synchronize, $SYNCH) or it requests the $QIOW system service. This step is repeated for each I/O operation.

❹ When the image has completed its I/O operations, it requests the $DASSGN system service. This service can be requested explicitly, or implicitly as part of image rundown or process deletion.

❺ If necessary, it explicitly requests the $DALLOC system service. This service can also be requested implicitly as part of image rundown or process deletion.

An I/O request is processed in a number of steps and threads of execution. A typical sequence is shown, with some simplifications, in Figure 23–2; the numbers in the figure correspond to those in the following list:

❶ The image requests the $QIO[W] system service.

❷ EXE$QIO, the $QIO system service procedure, runs in process context. It validates its device-independent arguments and builds a data structure, called an I/O request packet (IRP), that describes the I/O request. Although it is entered at interrupt priority level (IPL) 0, it raises IPL to 2 before it allocates the IRP.

❸ It invokes one or more function decision table (FDT) action routines specific to the device and I/O function. The FDT action routines, also running in process context, complete argument validation and any necessary I/O request preprocessing. An FDT routine can allocate a nonpaged pool buffer for use by the driver, and it can lock user buffer pages into memory so that they can be accessed by a direct memory access (DMA) device.

❹ In general, the last FDT action routine invokes an executive routine to pass the IRP to the device driver and to return control to the user. (An FDT action routine can also abort or complete the I/O request and return control to the user.)

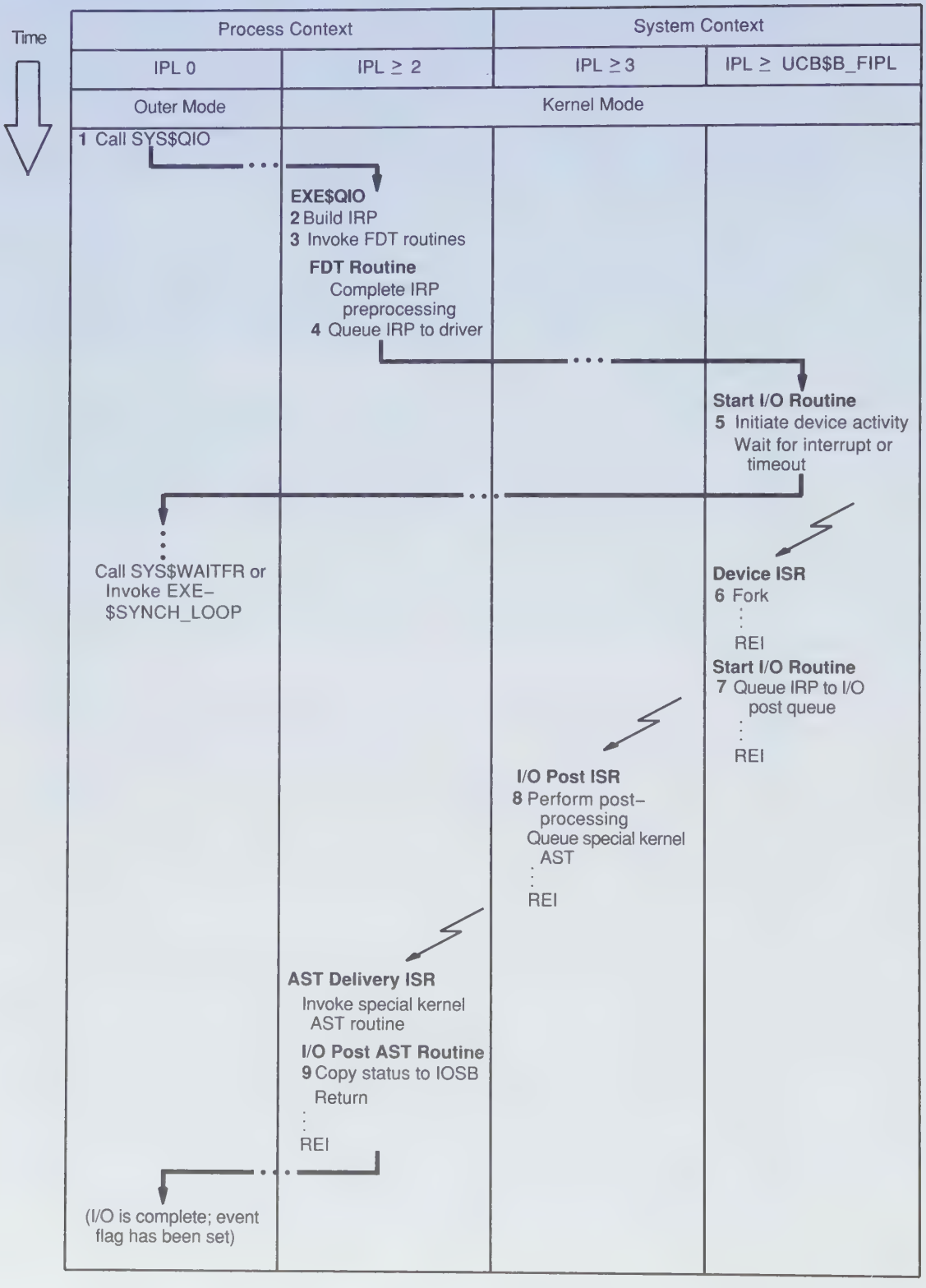
❺ The device driver's start I/O routine initiates the device activity corresponding to the I/O request and then waits for the device interrupt that signals completion of the activity. The start I/O routine can be entered in either process context or system context. Although this example describes a situation in which the routine would be entered in process context, for simplicity Figure 23–2 shows the start I/O routine in the System Context column.

❻ The device interrupt service routine (ISR), which executes at device IPL, copies device status and then forks to dismiss the interrupt and resume the start I/O routine at a lower IPL.

❼ Reentered as a fork process, the start I/O routine verifies that the request has been satisfied, copies status to the IRP, and then queues the IRP for postprocessing.

❽ The I/O postprocessing interrupt service routine, running in system context, performs some postprocessing functions, for example, unlock buffer pages, restore charged quota, and set the event flag associated with the I/O request. It queues a special kernel mode asynchronous system trap (AST) to the process whose I/O completed.

Figure 23–2 Flow of an I/O Request



9 Running in the context of the process that requested the I/O, the special kernel mode AST routine can copy I/O status to the image's I/O status block (IOSB) and copy input data from a nonpaged pool buffer to a user buffer. If the user requested AST notification of the I/O completion, the special kernel mode AST routine queues a normal AST.

23.2 Device Drivers and Spinlocks

[*Censored*] VMS Version 1 does not support symmetric multiprocessing. However, since [*censored*] VMS Version 1 is based on VAX/VMS Version 5.4, [*censored*] VMS drivers synchronize access to I/O devices and controllers with fork locks and device locks, both of which are types of spinlocks.

The effect of the FORKLOCK macro on a uniprocessing system is simply to raise IPL to the fork IPL for the device. The fork IPL is determined from the device's fork lock index in UCB$B_FLCK. Similarly, the effect of the DEVICELOCK macro on a uniprocessing system is to raise IPL to the device IPL, which is stored in UCB$B_DIPL by the device driver during its initialization. Chapter *Synchronization Techniques* discusses fork locks and device locks in more detail.

23.3 Device Attributes

The term *device* is defined in Chapter *I/O Architecture*.

The VMS executive assigns every device one or more attributes that describe the device to the I/O system services and the rest of the system. Table 23–1 shows a selected list of device attribute flags as defined by the STARLET $DEVDEF macro. These flags, which represent the characteristics of a device, are in the unit control block (UCB) fields UCB$L_DEVCHAR and UCB$L_DEVCHAR2.

A device may or may not have a physical embodiment. A device without a physical embodiment is called a pseudo device; it is a software emulation.

The bit UCB$V_TEMPLATE in UCB$L_STS, when set, indicates a template device. A template device, discussed in Section 23.5.2.2.2, is a pseudo device whose UCB is cloned whenever an image assigns a channel to it.

Table 23–1 Device Attribute Flags

Flag	Meaning if Set
DEV$V_REC	Device is record-oriented
DEV$V_CCL	Device supports carriage control
DEV$V_TRM	Device is a terminal
DEV$V_DIR	Device is directory-structured
DEV$V_SQD	Device is sequential and block-oriented
DEV$V_SPL	Device is being spooled
DEV$V_NET	Device is a network device
DEV$V_FOD	Device is file-oriented
DEV$V_DUA	Device is dual-ported
DEV$V_SHR	Device is shareable
DEV$V_AVL	Device is available for use
DEV$V_MNT	Device is mounted
DEV$V_MBX	Device is a mailbox
DEV$V_DMT	Device is marked for dismount
DEV$V_ELG	Device has error logging enabled
DEV$V_ALL	Device is allocated
DEV$V_FOR	Device is mounted foreign
DEV$V_IDV	Device is capable of providing input
DEV$V_ODV	Device is capable of providing output
DEV$V_DET	Device is a detached terminal
DEV$V_RED	Device is a redirected terminal

23.4 Allocating and Deallocating Devices

An I/O device can be declared shareable or nonshareable by its driver. A disk, for example, is typically declared shareable so as to allow concurrent access to multiple users, whereas a line printer is always declared nonshareable because access to it must be serialized. A device is typically declared nonshareable if its I/O is inherently sequential; such a device is designed to service I/O requests from one user at a time. VMS allows multiple processes to access a shareable device concurrently; under very constrained conditions, it is also possible for multiple processes to access a nonshareable device concurrently.

Before a process can issue I/O requests to a nonshareable device, it must allocate the device for its exclusive use. A process can also allocate a device that is declared shareable, thus temporarily acquiring exclusive access to it.

A device allocated to a process can nonetheless be used by another process under the following two conditions:

- The other process is a subprocess of the first. This condition provides, for example, flexible access to an interactive terminal among a user's process and its spawned subprocesses.

- The other process has the SHARE privilege. For example, the print symbiont uses the SHARE privilege to access a disk mounted privately when the owner queues files on the disk for printing.

In either set of circumstances, the processes sharing the device are responsible for arbitrating their accesses to it.

There are two forms of device allocation: explicit, requested by the process through the $ALLOC system service, and implicit, performed as necessary on behalf of the process by the $ASSIGN system service. In either form, the process ID (PID) of the process that allocated the device is stored in the UCB device owner field, UCB$L_PID.

Explicit allocation differs from implicit allocation in several ways:

- An implicitly allocated device is transparently deallocated when its last channel is deassigned; an explicitly allocated device must be explicitly deallocated.
- A process can request explicit allocation of a generic device type or a specific device unit.
- In the case of explicit allocation, the device-allocated bit, DEV$V_ALL in UCB$L_DEVCHAR, is set and the device reference count, UCB$W_REFC, is incremented twice, once by $ALLOC and once by $ASSIGN. In the case of implicit device allocation, the device-allocated bit is clear and the device reference count is incremented only once, by $ASSIGN.

A process requests the $ALLOC system service to allocate a device explicitly. The device can be released only through the $DALLOC system service, requested by the process directly or by code running on its behalf at image rundown or process deletion.

For a nonshareable device that a process has not explicitly allocated, the $ASSIGN system service (see Section 23.5.2) performs implicit allocation.

23.4.1 $ALLOC System Service

The $ALLOC system service has five arguments, of which only the DEVNAM argument is required:

- The DEVNAM argument identifies the device to be allocated.
- The PHYBUF argument specifies where the $ALLOC system service should return the name of the device.
- The PHYLEN argument specifies where it should return the length of the device name.
- The ACMODE argument identifies the access mode to be associated with the device. It is maximized with the mode of the requestor. Once allocated, the device can only be deallocated from the same or a more privileged mode.
- The FLAGS argument contains only one flag, the low bit. When set, the low bit indicates that any device of a particular type can be allocated, not just a specific device.

The $ALLOC system service procedure, EXE$ALLOC in module SYSDEVALC, will not allocate the device if any one of the following conditions is true:

- The device is already allocated by another process (UCB$L_PID is nonzero and does not match PCB$L_PID).
- The device reference count is nonzero.
- A volume is mounted on the device.
- The device is spooled (DEV$V_SPL in UCB$L_DEVCHAR is set), and the process does not have the ALLSPOOL privilege.
- The requesting process does not have access rights to allocate the device, based on the device owner's user identification code (UIC) and protection (fields UCB$L_UIC and UCB$W_PROT) and its access control list.
- The device is not available (DEV$V_AVL in UCB$L_DEVCHAR is clear) or not online (UCB$V_ONLINE in UCB$L_STS is clear).
- The device is a template device.

EXE$ALLOC runs in kernel mode. It takes the following steps to allocate a device:

1. It locks the I/O database mutex for write access.
2. It verifies that the DEVNAM argument's string descriptor is read-accessible.
3. If the FLAGS argument is specified, EXE$ALLOC verifies that it is read-accessible and does not have undefined bits set.
4. It invokes IOC$SEARCH, in module IOSUBPAGD, to locate a suitable device.
   - — If the FLAGS argument is not specified or is 0, EXE$ALLOC requests a search for the exact device specified by the DEVNAM argument.
   - — If the FLAGS argument is 1, EXE$ALLOC requests a search for the first available device matching the type specified by the DEVNAM argument.

   IOC$SEARCH invokes IOC$TRANDEVNAM, in module IOSUBPAGD, to translate the DEVNAM argument. It then searches the I/O database for either the specific device or one of the particular type. IOC$SEARCH and routines it invokes verify the suitability of the device and its accessibility to this process.
5. EXE$ALLOC returns the translated device name if the PHYBUF argument is specified, the descriptor is readable, and the buffer is writable. If the PHYLEN argument is also specified and is write-accessible, EXE$ALLOC also returns the length of the device name.
6. It allocates the device:
   a. It sets the device-allocated bit, DEV$V_ALL in UCB$L_DEVCHAR.
   b. It maximizes the ACMODE argument with the access mode of its requestor and stores the result in UCB$B_AMOD.
   c. It increments the device reference count, UCB$W_REFCNT.
   d. It copies the process ID, PCB$L_PID, to the UCB device owner field, UCB$L_PID.

7. It unlocks the I/O database mutex by invoking IOC$UNLOCK, in module IOSUBPAGD, and returns to the requestor with the success status SS$_NORMAL.

23.4.2 $DALLOC System Service

An image can deallocate a single device or all devices allocated to the process by requesting the $DALLOC system service. The $DALLOC system service is also requested by the Rundown ($RUNDWN) system service during image rundown to deallocate all user-mode devices and during process deletion to deallocate all devices still allocated to the process. The $RUNDWN system service is discussed in Chapter *Image Activation and Exit*. Process deletion is discussed in Chapter *Process Deletion*.

The $DALLOC system service has two optional arguments:

- The DEVNAM argument specifies the device to be deallocated. If the DEVNAM argument is specified, it must translate to a physical device name. If the DEVNAM argument is not specified, all devices allocated by the process from access modes equal to or less privileged than that specified by the ACMODE argument are deallocated.
- The ACMODE argument specifies the access mode on whose behalf the deallocation is to be performed. It is maximized with the mode of the requestor.

The $DALLOC system service procedure, EXE$DALLOC in module SYSDEVALC, runs in kernel mode. It performs the following steps:

1. It maximizes the ACMODE argument with the access mode of its requestor.
2. It locks the I/O database mutex for write access.
3. It determines if the DEVNAM argument is present.
   - — If the argument is present, it invokes IOC$SEARCHDEV, in module IOSUBPAGD, to locate the specified device.
   - — If the argument is absent, it invokes IOC$SCAN_IODB, in module IOSUBNPAG, to find the first UCB in the I/O database.
4. In either case, EXE$DALLOC makes the following checks before deallocating the device.
   - — The UCB$L_PID field must match the PCB$L_PID field of the process requesting the $DALLOC system service.
   - — The access mode in UCB$B_AMOD must be greater than or equal to the access mode computed in step 1.
   - — The device must have been explicitly allocated.
   - — The device must not be mounted (DEV$V_MNT in UCB$L_DEVCHAR must be clear) unless the device is a terminal (DEV$V_TRM in UCB$L_DEVCHAR is set). A DECnet remote terminal is marked as mounted but need not be interlocked against deallocation because only the owner process has access to it.

5. It deallocates the device by invoking IOC$DALLOC_DEV, in module IOSUB-PAGD, which takes the following steps:
   a. It clears the device-allocated bit.
   b. If the device is shareable, it clears the device owner field.
   c. It decrements the device reference count.
   d. If the reference count is now zero, IOC$DALLOC_DEV clears the owner field in the UCB and invokes IOC$LAST_CHAN, which performs last channel processing (see Section 23.5.4).
6. If the DEVNAM argument was present, EXE$DALLOC is done. It unlocks the I/O database mutex by invoking IOC$UNLOCK and returns to its requestor with the success status SS$_NORMAL.

   Otherwise, EXE$DALLOC goes to step 3 to get the next UCB in the I/O database. When no more UCBs are found, EXE$DALLOC is done and exits as described.

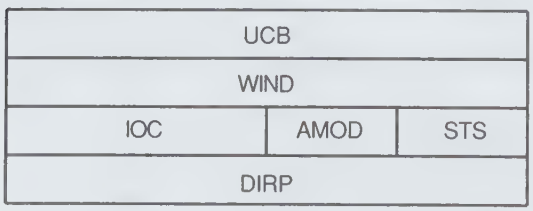
23.5 Assigning and Deassigning Channels

The software mechanism that links a process to a device is called a channel. To perform I/O on a device, an image first creates a channel to it by requesting the $ASSIGN system service. The image then identifies the device to the $QIO system service through its channel number. When the image is done with the device, it requests the $DASSGN system service to break the link between the process and the device.

23.5.1 Channel Control Block

A channel is described by a process-specific data structure called a channel control block (CCB). A process's CCBs are contained in a table located in its P1 space. The global location CTL$GL_CCBBASE contains the address of the table's high-address end. The table is accessed using negative byte displacements. That is, a particular CCB is identified by its displacement from the contents of CTL$GL_CCBBASE. The number of CCBs in the table is determined by the SYSGEN parameter CHANNELCNT. Figure 23–3 shows the layout of a CCB.

Figure 23–3 Layout of a Channel Control Block



The field CCB$B_AMOD contains 0 if the channel is unassigned. Otherwise, it contains the access mode from which the channel was assigned, biased by 1. For example, the value 1 indicates that the channel was assigned from kernel mode. A $QIO system service request on a particular channel must be made from an access mode at least as privileged as the mode from which the channel was assigned.

CCB$L_UCB contains the address of the UCB of the device to which the channel is assigned.

Any comparison of CCB$B_AMOD with an access mode value must be a signed comparison. The Files-11 XQP prevents deassignment of its channel when the channel is inactive by storing –1 in CCB$B_AMOD. (The XQP reserves this channel for itself during process initialization.) Prior to using the channel, the XQP transforms the CCB into a normal kernel mode channel to the device of the XQP's choice.

If a file has been opened on the channel, CCB$L_WIND contains the address of its window control block (WCB). If the file is associated with a process section, CCB$L_WIND contains the process section index. CCB$L_WIND contains an unnamed flag in the low bit that is set to indicate either an access (open) request in progress or a deaccess (close) request waiting for all other outstanding I/O requests to be completed.

- If an access request is pending, CCB$L_WIND contains a 1.
- If a deaccess request is pending, CCB$L_WIND contains the result of a logical OR operation of the WCB address or process section index with 1. CCB$L_DIRP contains the address of the IRP that describes the deaccess request. Since WCB addresses and process section indexes are always even, system routines can recover these values by masking out the low bit of CCB$L_WIND.

CCB$B_STS contains several status bits.

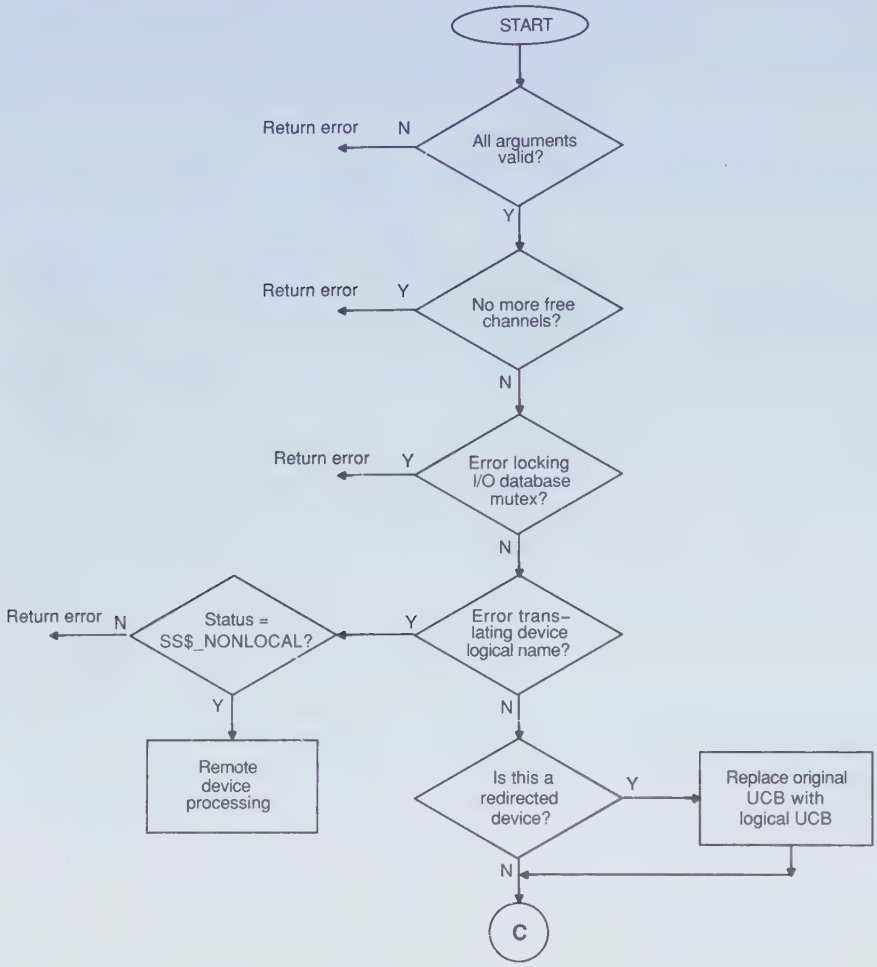
The field CCB$W_IOC indicates how many I/O requests are outstanding on the channel.

23.5.2 $ASSIGN System Service

The $ASSIGN system service has four arguments; the first two are required and the last two are optional:

- The DEVNAM argument is the name of the device to which to assign the channel.
- The CHAN argument is the address of the word in which to return the assigned channel number.
- The ACMODE argument, indicating the access mode to be associated with the channel, is maximized with the mode of the requestor.
- The MBXNAM argument is the name of the mailbox to be associated with the channel. An image associates a mailbox with a nonshareable device to receive status information, such as the arrival of unsolicited input from a terminal. The device driver for the device either uses or ignores this associated mailbox.

Figure 23–4 Major Steps in EXE$ASSIGN_LOCAL—Part 1

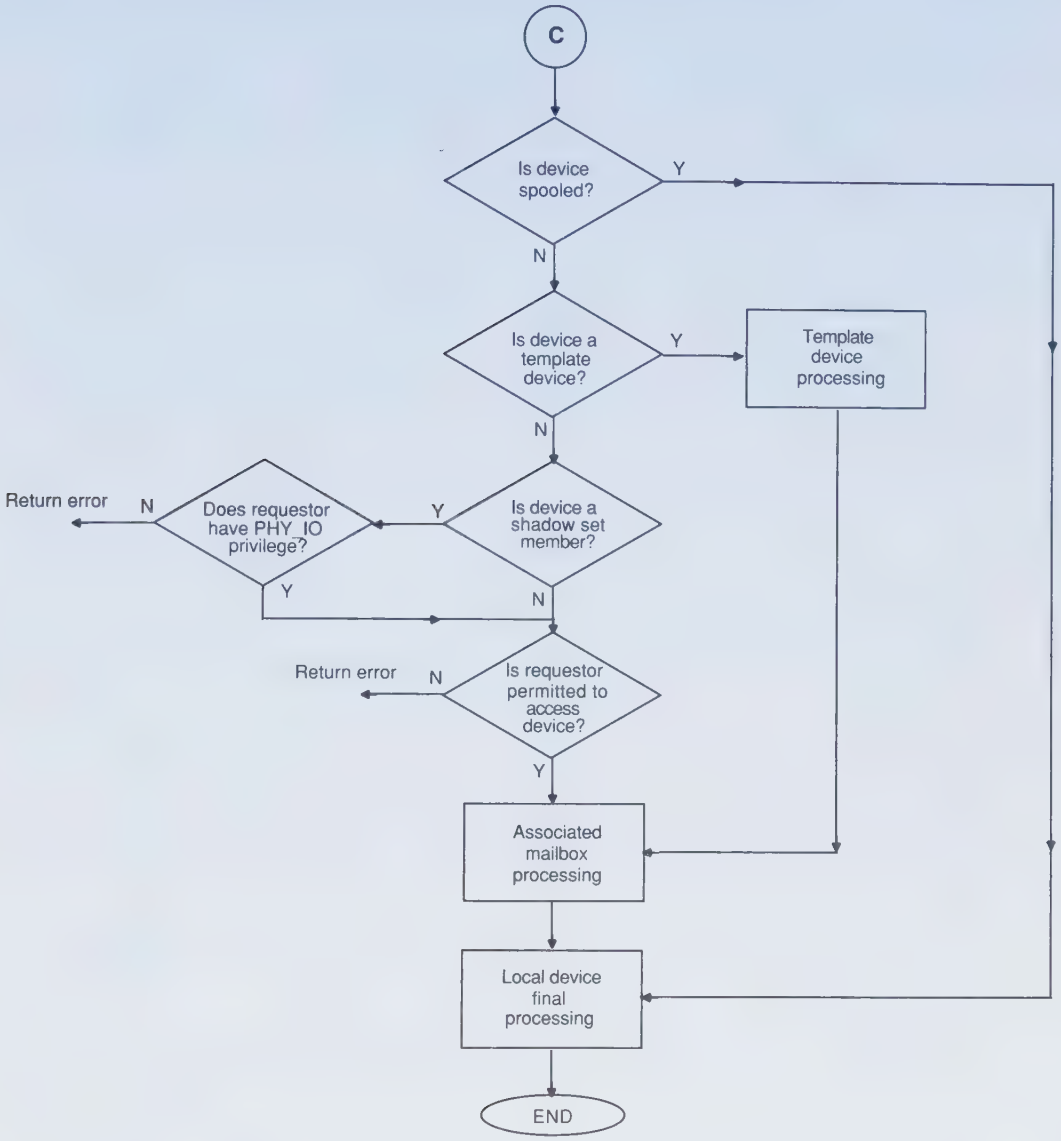


The $ASSIGN system service procedure, EXE$ASSIGN in module SYSASSIGN, performs special processing for DECnet remote devices. A DECnet remote device is one whose device name specification includes the string :: . A device that is not a DECnet remote device is, for the purposes of the $ASSIGN system service, called a local device. Note that even a local device can be external to the system; the device's driver can hide the details of device implementation from the $ASSIGN system service. A local area terminal (LAT) device, for example, is a local device by this definition, even though the physical terminal associated with such a device can be connected to the system through a network.

EXE$ASSIGN runs in the caller's mode. It performs the following steps:

1. It requests the $ASSIGN_LOCAL system service, passing it the same arguments as listed for $ASSIGN. EXE$ASSIGN_LOCAL, in module SYSASSIGN, is the $ASSIGN_LOCAL system service procedure. EXE$ASSIGN_LOCAL assigns a channel to a local device; it returns the status SS$_NONLOCAL for a device whose name has the string :: embedded in it. The actions of EXE$ASSIGN_LOCAL are detailed in the next few sections.

Figure 23–5 Major Steps in EXE$ASSIGN_LOCAL—Part 2



2. If EXE$ASSIGN_LOCAL returns the success status SS$_NONLOCAL, EXE$ASSIGN calls NETWORK_ASSIGN, also in module SYSASSIGN, to assign a channel to a DECnet remote device.
3. EXE$ASSIGN returns to its requestor.

Figures 23–4 and 23–5 show the major steps in the control flow of EXE$ASSIGN_LOCAL.

23.5.2.1 Common Initial Steps

Running in kernel mode, EXE$ASSIGN_LOCAL performs the following steps for both local and DECnet remote device assignment:

1. It verifies that the CHAN argument is write-accessible.
2. If the MBXNAM argument was specified, EXE$ASSIGN_LOCAL verifies that it is read-accessible.
3. It verifies that the DEVNAM argument is read-accessible.
4. It verifies that the ACMODE argument is read-accessible and maximizes the argument with the access mode of its requestor.
5. It invokes IOC$FFCHAN, in module IOSUBPAGD, to find a free CCB.

   IOC$FFCHAN begins its search for a free CCB at the high-address end of the CCB table. It examines offset CCB$B_AMOD to determine whether the CCB is free. If the CCB is in use, IOC$FFCHAN examines the next CCB, repeating its test. This sequence continues until IOC$FFCHAN locates a free CCB or reaches the end of the table.

   If IOC$FFCHAN locates a CCB, it returns the address of the free CCB and a positive offset into the CCB table. This offset is the channel number returned from the system service request.

   If no free CCB is located, IOC$FFCHAN returns the error status SS$_NOIOCHN.
6. EXE$ASSIGN_LOCAL locks the I/O database mutex for write access.
7. If the MBXNAM argument was specified, EXE$ASSIGN_LOCAL invokes IOC$SEARCHDEV to get the address of the specified mailbox UCB. The device must be a mailbox device (DEV$V_MBX in UCB$L_DEVCHAR is set) but not a network device.
8. It invokes IOC$SEARCH to locate the device specified in the DEVNAM argument. If the device name is a logical name, IOC$SEARCH invokes IOC$TRANDEVNAM to perform logical name translation.

   If IOC$TRANDEVNAM returns a success status, IOC$SEARCH then scans the I/O database for a device with the resulting equivalence name.

   — If IOC$SEARCH locates the device, it returns the address of the device's UCB. EXE$ASSIGN_LOCAL then takes the steps discussed in Section 23.5.2.2.

   — If IOC$SEARCH does not locate the device, EXE$ASSIGN_LOCAL unlocks the I/O database mutex by invoking IOC$UNLOCK and returns to the requestor with the error status from IOC$SEARCH.
9. If the device name contains a node delimiter (::), IOC$TRANDEVNAM returns the error status SS$_NONLOCAL, and EXE$ASSIGN_LOCAL takes the steps described in Section 23.5.2.3 for DECnet remote device assignment.
10. If IOC$TRANDEVNAM returns any other error status, EXE$ASSIGN_LOCAL unlocks the I/O database mutex by invoking IOC$UNLOCK and returns to the requestor with that error status.

23.5.2.2 Local Device Assignment

EXE$ASSIGN_LOCAL first checks for several special kinds of device:

1. If the UCB is a redirected UCB (DEV$V_RED in UCB$L_DEVCHAR2 is set), EXE$ASSIGN_LOCAL replaces the original UCB address with the address of the logical UCB by using the value in field UCB$L_TT_LOGUCB of the original UCB. This mechanism associates the assigned channel with the virtual terminal rather than with the physical. (The physical terminal can be a pseudo device such as a LAT terminal.) Only terminal UCBs can be redirected. Chapter *Miscellaneous I/O Topics* gives more details.
2. If the device is set spooled, EXE$ASSIGN_LOCAL goes directly to local device final processing, described in Section 23.5.2.2.4.

EXE$ASSIGN_LOCAL then determines whether the device is a template device. If it is, EXE$ASSIGN_LOCAL clones the UCB to create a new device, as described in Section 23.5.2.2.2.

23.5.2.2.1 Nontemplate Device Processing

Before assigning a channel to a local nontemplate device, EXE$ASSIGN_LOCAL confirms the following:

- If the device is a shadow set member (DEV$V_SSM or DEV$V_SHD in UCB$L_DEVCHAR2 is set), the requesting process must have the PHY_IO privilege.
- If the device is allocated (UCB$L_PID is nonzero) and nonshareable, one of the following two conditions must be true:
  - The requesting process must be the owner of the device or a descendant of the owner process.
  - The requesting process must have the SHARE privilege, and the volume protection and owner UIC must allow access.
- If the device is not allocated, the volume protection and owner UIC must allow access.

If the requestor is allowed to assign a channel to the device, EXE$ASSIGN_LOCAL handles the associated mailbox, if any, and performs final processing (see Section 23.5.2.2.4).

If the requestor is not allowed to assign the channel, EXE$ASSIGN_LOCAL unlocks the I/O database mutex by invoking IOC$UNLOCK and returns to the requestor with an error status.

23.5.2.2.2 Template Device Processing

If the device is a template device, EXE$ASSIGN_LOCAL creates a new UCB, called the cloned UCB, by copying the template UCB, and assigns the channel to the cloned UCB as follows:

1. If the template device is a network device, it verifies that the process has NETMBX privilege.

2. EXE$ASSIGN_LOCAL invokes IOC$CHKUCBQUOTA, in module UCBCREDEL, to verify that the process has as much BYTLM quota as the sum of the size of the template UCB plus 256 additional bytes to satisfy process deletion needs.

   IOC$CHKUCBQUOTA invokes EXE$DEBIT_BYTCNT_BYTLIM_NW, in module EXSUBROUT, to check and charge the quota. Both the byte count quota (JIB$L_BYTCNT) and limit (JIB$L_BYTLIM) are charged. Since the amount charged by IOC$CHKUCBQUOTA will not be restored until the UCB is deleted, the process has effectively had its byte limit reduced by the amount of the charge. EXE$DEBIT_BYTCNT_BYTLIM_NW decrements the byte limit as well to reflect this fact.

3. EXE$ASSIGN_LOCAL invokes IOC$CLONE_UCB, in module UCBCREDEL, to create the cloned UCB and an object rights block (ORB).

   IOC$CLONE_UCB copies the template UCB and then makes several modifications to the cloned UCB. The following are of particular interest:

   — It sets the reference count to 1.

   — It marks the unit online.

   — It clears the template bit.

   — It stores the size of the UCB in UCB$W_CHARGE.

   — It gives the UCB a unique unit number between 1 and 9999.

   — It links the UCB into the UCB chain of the related device data block (DDB).

4. EXE$ASSIGN_LOCAL stores the current process's UIC in the ORB owner field (ORB$L_OWNER). At this point, the owner field of the cloned UCB is still clear.

5. It sets UCB$V_DELETEUCB in UCB$L_STS to mark the cloned UCB for deletion when its reference count goes to 0.

6. If the template UCB is a mailbox, it sets the mailbox delete bit (UCB$V_DELMBX in UCB$W_DEVSTS). This is done because special steps are required to delete a mailbox UCB.

7. It clears the device reference count, which was set to 1 by IOC$CLONE_UCB. Later, when channel assignment is complete, the reference count is incremented.

8. EXE$ASSIGN_LOCAL invokes IOC$DEBIT_UCB, in module UCBCREDEL, to record the master PID charged for the UCB (JIB$L_MPID) into the charge PID field (UCB$L_CPID).

9. EXE$ASSIGN_LOCAL invokes the driver at the entry point specified by DDT$L_CLONEUCB, passing it the addresses of the template and cloned UCBs. The driver can perform any additional checks necessary. If the driver returns any error status, the process of cloning the UCB is undone and the $ASSIGN completes with failure.

The driver's cloned UCB routine runs in the context of the process that requested the $ASSIGN system service. It executes at IPL 2 because the I/O database mutex is owned by the process.

10. If the device is not shareable, EXE$ASSIGN_LOCAL copies the process's PID to UCB$L_PID, implicitly allocating it.

11. It takes the steps described in Section 23.5.2.2.3.

23.5.2.2.3 Associated Mailbox Processing

If an associated mailbox was requested, EXE$ASSIGN_LOCAL stores the address of the associated mailbox UCB in the UCB$L_AMB field of the UCB to which the channel is being assigned. It increments the reference count in the associated mailbox UCB and sets CCB$V_AMB for later storage in CCB$B_STS to indicate that there is an associated mailbox.

No association is made if either of the following is true:

- The device is a file-oriented device (DEV$V_FOD in UCB$L_DEVCHAR is set) or the device is shareable. In either case, the request for an associated mailbox is simply ignored.
- The device already has an associated mailbox (UCB$L_AMB is nonzero), and the MBXNAM argument specifies a different mailbox. In this case, EXE$ASSIGN_LOCAL unlocks the I/O database and returns the failure status SS$_DEVACTIVE.

Upon completing any steps required for the associated mailbox, EXE$ASSIGN_LOCAL proceeds to final processing (see Section 23.5.2.2.4).

23.5.2.2.4 Local Device Final Processing

At this point, EXE$ASSIGN_LOCAL has found a free channel, verified the existence of the device (creating the UCB in the case of a template device), and verified that the process has access to the device. It completes the assignment of an I/O channel to a local device in the following steps:

1. If the device is not shareable and not currently owned, EXE$ASSIGN_LOCAL implicitly allocates the device to the current process by storing the current process's PID (PCB$L_PID) in UCB$L_PID.
2. It copies the device's UCB address to CCB$L_UCB.
3. It increments the device reference count.
4. It stores the access mode biased by 1 in CCB$B_AMOD.
5. It sets CCB$B_STS appropriately. The only bit that can be set as a result of this step is CCB$V_AMB.
6. EXE$ASSIGN_LOCAL writes the channel number, the offset into the CCB table, in the word specified by the CHAN argument.
7. It unlocks the I/O database mutex by invoking IOC$UNLOCK and returns the success status SS$_NORMAL to its requestor.

23.5.2.3 Assigning a Channel to a DECnet Remote Device

This section assumes familiarity with transparent and nontransparent network communication, described in the *VMS Networking Manual*. If the device is a DECnet remote device, EXE$ASSIGN performs the first step in transparent network communication, converting the transparent network communication into the related nontransparent network communication.

If IOC$TRANDEVNAM returns the failure status SS$_NONLOCAL (see Section 23.5.2.1), EXE$ASSIGN_LOCAL converts it to a success status. EXE$ASSIGN_LOCAL then invokes IOC$UNLOCK to unlock the I/O database mutex and to return the now-modified SS$_NONLOCAL status to EXE$ASSIGN. In response to this, EXE$ASSIGN calls NETWORK_ASSIGN, in module SYS-ASSIGN.

NETWORK_ASSIGN, running in the access mode from which the $ASSIGN system service was requested, initiates nontransparent network communication by taking the following steps:

1. It establishes a condition handler in case system service failure exception mode has been enabled. This condition handler will resignal any conditions other than SS$_NOLOGNAM, which occurs normally in logical name translation and should not be passed back to the requestor of the $ASSIGN system service.
2. It allocates a buffer on the stack for use as the data area for logical name translation and initializes this area to request the equivalence name and its attributes.
3. It requests the Translate Logical Name ($TRNLNM) system service to translate the DEVNAM argument. This repetition of the logical name translation done at the beginning of EXE$ASSIGN_LOCAL is necessary because the result of the earlier translation was not saved.

   The result of this step should be a network connect block suitable for use in an outbound connection request operation. NETWORK_ASSIGN makes no attempt to ensure that the result of this step is in the proper format. If it is not, an error will be returned when the connection is attempted in the next step.
4. NETWORK_ASSIGN requests the $ASSIGN system service with the following items in the argument list:
   - The DEVNAM argument is the network device name, _NET.
   - The CHAN argument is a stack location that temporarily holds the assigned channel number.
   - The ACMODE argument is the ACMODE argument of the original $ASSIGN request, maximized with the access mode of the requestor.
   - The MBXNAM argument is the same argument passed in the original $ASSIGN system service request.

   Since NET0 is a template device, the unit to which the channel is assigned is a new unit, created as described in Section 23.5.2.2.2.

5. It requests the $QIOW system service to establish a connection to the DECnet remote device:
   - The FUNC argument is IO$_ACCESS ORed with IO$M_ACCESS.
   - The event flag is EXE$C_SYSEFN.
   - The CHAN argument is the one to which the device was assigned in the previous step.
   - The network connect block is the one obtained in step 3.
6. If the $QIOW completes successfully, NETWORK_ASSIGN records the channel number from step 4 in the word specified by the CHAN argument of the original $ASSIGN system service request. It then returns the success status SS$_REMOTE to its requestor.
7. If the $QIOW fails, NETWORK_ASSIGN requests the $DASSGN system service to deassign the channel. It then returns the failure status from the $QIOW system service to its requestor.

23.5.3 $DASSGN System Service

The $DASSGN system service deassigns a previously assigned I/O channel and clears the linkage and control information in the corresponding CCB, freeing the CCB for reuse. Any outstanding I/O request on the device is terminated in the process. $DASSGN has only one argument, the CHAN argument, which specifies the channel to be deassigned.

The $DASSGN system service procedure, EXE$DASSGN in module SYSDASSGN, runs in kernel mode. It takes the following steps:

1. It invokes IOC$VERIFYCHAN, in module IOSUBPAGD, which performs the following steps:
   a. It verifies that the channel is legal.
   b. It verifies that the channel was assigned from an access mode no more privileged than the access mode from which it is to be deassigned. CCB$B_AMOD must be greater than the previous mode field in the processor status (PS).
   c. It returns the address and the index of the CCB for the channel.
2. EXE$DASSGN calls EXE$CANCELN with a reason code of CAN$C_DASSGN (channel is being deassigned) to cancel all outstanding I/O on the channel. EXE$CANCELN is an entry point in the $CANCEL system service, discussed in Section 23.9.
3. It invokes IOC$VERIFYCHAN again in case the cancel I/O operation triggered a kernel mode AST routine that requested the $DASSGN system service again. This second call to $DASSGN could have completely deassigned the channel.
4. If a file is open on the channel (CCB$L_WIND is nonzero), EXE$DASSGN requests the $QIOW system service to close the file. It specifies a function code of IO$_DEACCESS and event flag number 30. Event flag 30 is used to avoid conflict with the use of event flag 31 (EXE$C_SYSEFN) by $CANCEL.

If a network logical link rather than a file was open on the channel, the deaccess operation dissolves the link.

5. EXE$DASSGN raises IPL to 2 and examines CCB$W_IOC to determine if any I/O is outstanding on the channel. If there is, EXE$DASSGN must wait for its completion AST before proceeding further. EXE$DASSGN acquires the SCHED spinlock and tests whether the process has a pending kernel mode AST whose delivery has been blocked by EXE$DASSGN's execution at IPL 2 and above.

   — If there is a pending kernel mode AST, EXE$DASSGN releases the spinlock, lowers IPL to 0, and transfers control to step 3. (Lowering IPL to 0 triggers the interrupt that delivers the pending kernel mode AST.)

   — If there is not, EXE$DASSGN calls SCH$RESOURCE_WAIT_PS, in module MUTEX, to place the process into a resource wait. SCH$RESOURCE_WAIT_PS releases the SCHED spinlock and waits the process at IPL 0 and at a PC corresponding to step 3.

   Chapter *ASTs* discusses ASTs in more detail, and Chapter *Scheduling*, wait states.

6. It locks the I/O database mutex for write access.
7. It clears CCB$B_AMOD.
8. If there is an associated mailbox (CCB$V_AMB in CCB$B_STS is set), EXE$DASSGN dissociates the mailbox by taking the following steps:
   a. It clears UCB$L_AMB in the device UCB.
   b. It decrements the reference count in the mailbox UCB.
   c. If the mailbox reference count is now 0, it invokes IOC$LAST_CHAN_AMBX, in module IOSUBNPAG, to perform last channel processing for an associated mailbox (see Section 23.5.4).
9. It decrements the reference count in the device UCB.
10. If the device reference count is now 0, indicating that the device was not explicitly allocated, EXE$DASSGN takes the following steps:
    a. It clears the device owner field, deallocating the device.
    b. It invokes IOC$LAST_CHAN to perform last channel processing.
11. If the device reference count is 1 and the device has been explicitly allocated, EXE$DASSGN invokes IOC$LAST_CHAN to perform last channel processing.
12. It invokes IOC$UNLOCK to unlock the I/O database mutex and returns the success status SS$_NORMAL to its requestor.

23.5.4 Last Channel Processing

Last channel processing is performed when the last channel to a device is deassigned:

- When the device reference count goes to 0, and the device was not explicitly allocated
- When the device reference count goes to 1, and the device was explicitly allocated

There are two entry points to last channel processing: IOC$LAST_CHAN and IOC$LAST_CHAN_AMBX. The latter routine is invoked when the device is an associated mailbox, the former routine in all other cases. They differ only in their initial steps:

- IOC$LAST_CHAN is invoked with the channel number and the address of the UCB of the device assigned to the channel. It saves the reason code CAN$C_DASSGN for later use.
- IOC$LAST_CHAN_AMBX is invoked with the address of the mailbox UCB, not the UCB of the device assigned to the channel. (The channel is not assigned to the mailbox and is not needed by the mailbox driver. The current IRP is also not needed by the mailbox driver.) It saves the reason code CAN$C_AMBXDGN for later use.

At this point, IOC$LAST_CHAN and IOC$LAST_CHAN_AMBX converge in the following steps:

1. The routine acquires the device fork lock, raising IPL to the associated fork IPL. This step synchronizes access to the UCB.
2. It invokes the device driver's cancel I/O routine, passing the reason code saved previously.
3. The routine releases the fork lock without changing IPL.
4. It lowers IPL to 2, leaving it there to prevent process deletion.
5. If the device is explicitly allocated, the routine returns to its invoker.
6. If the device is a terminal or mailbox, the routine clears DEV$V_OPR in UCB$L_DEVCHAR, disabling the device as an operator terminal.
7. If UCB$V_DELETEUCB in UCB$L_STS is set, the routine takes the following two steps:
   a. It invokes IOC$CREDIT_UCB, in module UCBCREDEL, to return the quota charged against the byte count and byte limit.
   b. It invokes IOC$DELETE_UCB, in module UCBCREDEL, to delete the UCB and the associated ORB.
8. The routine returns to its invoker.

23.6 $QIO System Service

The $QIO system service performs device-independent preprocessing and, via FDT routines, device-dependent preprocessing. It then queues an I/O request to the driver for the device associated with a channel. Any additional work to be done is performed by the device driver's start I/O routine.

The $QIO system service has the following arguments:

- The EFN argument is the number of the event flag to be associated with the I/O request. Since this argument is passed by value, omitting it is the same as specifying event flag 0.
- The CHAN argument is the number of the I/O channel. This is the same as the CHAN argument returned by the $ASSIGN system service.
- The FUNC argument identifies what operation is to be performed by the device driver. It is divided into two portions, the function code proper and function modifiers. Throughout the chapter, the term *function code* means just the function code proper; the term FUNC means the entire argument.
- The IOSB argument is the address of the IOSB, a quadword to receive final status of the I/O operation. The *VMS System Services Reference Manual* and the *VMS I/O User's Reference Volume* contain detailed descriptions of the format of the IOSB.
- The ASTADR argument is the address of an AST procedure to be executed in the mode of the requestor when the I/O operation completes.
- The ASTPRM argument is the parameter to be passed to the AST procedure.
- There are six optional device- and function-specific parameters, P1 through P6.

The CHAN and FUNC arguments must be specified. All others are optional and, if not specified, default to a value of zero.

23.6.1 Device-Independent Preprocessing

The $QIO system service procedure, EXE$QIO in module SYSQIOREQ, executes in kernel mode.

To perform device-independent preprocessing, EXE$QIO validates and processes all its arguments except for P1 through P6. It takes the following steps:

1. It clears the specified event flag so that the process will be placed into a wait state until the I/O operation completes, should the caller request either the $SYNCH system service or one of the event flag wait system services to wait for the I/O operation to complete.
2. It verifies that the channel number is valid and has been assigned from an access mode no more privileged than the mode of the $QIO requestor by performing the following checks:
   — The channel number is greater than zero and less than or equal to the contents of CTL$GW_CHINDX. CTL$GW_CHINDX contains the number of the highest assigned channel. Note that not all the channels whose numbers are less than the contents of CTL$GW_CHINDX are necessarily

currently assigned. They could have been deassigned since the channel whose number is stored in CTL$GW_CHINDX was last assigned.

— The access mode of the requestor (specified by the previous mode field, PSL$V_PRVMOD, of the current PS) is less than the access mode specified by the CCB access mode field. This ensures that the channel is used only from access modes at least as privileged as the access mode from which the channel was assigned.

3. If an access or deaccess request is pending on the channel (low bit in CCB$L_WIND is set), the process is placed in an AST wait state, to wait for the access or deaccess to complete. When the AST wait is satisfied, EXE$QIO will restart at the beginning.

4. It extracts the function code from the FUNC argument.

5. If the device is spooled and the function code specifies a virtual I/O function, EXE$QIO substitutes the intermediate device UCB for the UCB specified in the CCB. The intermediate device UCB address is stored in UCB$L_AMB of the UCB specified by the CCB. Virtual I/O to a spooled device is assumed to be I/O that should be spooled. I/O done by the software implementing spooling, for example, the print symbiont, would be logical or physical I/O.

6. Under some circumstances, EXE$QIO must verify the process's access to the device. If the device is file-oriented, then a file processor (ACP or Files-11 XQP) has been or will be involved in checking the process's access to the device when it opens a file. If the device is neither file-oriented nor shareable, the process's access has already been checked as part of implicit or explicit device allocation.

   However, when a process requests a read or write operation from a shareable, non-file-oriented device (for example, a real-time device or one mounted foreign), EXE$QIO checks whether the access is allowed. It invokes either EXE$CHKRDACCES or EXE$CHKWRTACCES, in module EXSUBROUT. If the process has the needed access, the routine sets the appropriate bit (CCB$V_RDCHKDON or CCB$V_WRTCHKDON) in CCB$B_STS.

   Note that EXE$QIO contains two lists of functions, one for reads and one for writes. While the interpretation of function codes is almost entirely up to the device driver, EXE$QIO does know that the "correct" interpretation of certain codes is a read or a write operation and performs access checking based on this interpretation.

   In step 16, EXE$QIO performs additional access checks based on whether the I/O function is physical, logical, or virtual.

7. EXE$QIO verifies that the function code is a legal function by checking the legal function mask in the FDT (see Chapter *Overview of the I/O Subsystem*).

8. If the device is offline, EXE$QIO checks that the function code is either IO$_DEACCESS or IO$_ACPCONTROL. If it is not, EXE$QIO returns the error status SS$_DEVOFFLINE.

9. If the IOSB argument is nonzero, EXE$QIO verifies that the IOSB can be written by the requesting mode and then clears it.

10. EXE$QIO uses the buffered I/O function mask in the FDT to determine whether the function code specifies a direct or buffered operation.

11. It raises IPL to 2 to prevent process deletion. This step is necessary for two reasons:

    — EXE$QIO will allocate an IRP. The fact that this IRP is allocated to this process will not be reflected in any data structure until much later. If the process were to be deleted before this allocation were recorded, the IRP would be lost.

    — In steps 12 and 14, EXE$QIO indicates that this process has outstanding I/O. If process deletion were begun after these steps, but before the request was actually queued, the process would become deadlocked, trying to run down nonexistent I/O. For more information on process deletion, see Chapter *Process Deletion*.

12. It determines whether the process has sufficient I/O quota (direct or buffered, depending upon the previous determination) and, if so, charges against it.

    If the process does not have sufficient quota, EXE$QIO invokes EXE$SNGLQUOTA, in module EXSUBROUT, to place the process in an AST wait if the process has resource wait mode enabled.

13. It allocates an IRP from nonpaged pool. Chapter *Pool Management* discusses IRP allocation.

14. It increments the outstanding I/O count in the CCB.

15. It initializes the IRP. Most of this initialization is straightforward, for example, storing the EFN argument in IRP$B_EFN. There are some steps that deserve special comment:

    — If the ASTADR argument is nonzero, EXE$QIO charges the process AST quota for an AST control block (ACB). It also sets ACB$V_QUOTA in IRP$B_RMOD to indicate that the process has been charged for the ACB.

    — If the function code specifies a buffered I/O operation, EXE$QIO sets IRP$V_BUFIO in IRP$W_STS. Otherwise, it clears the bit.

    — EXE$QIO clears the fields that describe the buffer, IRPL_SVAPTE, IRPW_BOFF, and IRP$L_BCNT, the transfer parameters.

    — If CCB$L_WIND is nonzero, the channel is associated with either a file or a process section. If the channel is associated with a file, CCB$L_WIND contains the system space address of a WCB, a negative number. EXE$QIO stores the address of this WCB in IRP$L_WIND.

      If the channel is associated with a process section, CCB$L_WIND contains the process section index, a positive number. EXE$QIO uses this value to index the process section table (PST) and obtain the address of the WCB associated with the process section. EXE$QIO stores the address of this WCB in IRP$L_WIND. Chapter *Memory Management Data Structures* contains details on the PST.

— If the function code is a virtual read or write to a non-file-oriented device, EXE$QIO converts the function code into the corresponding logical function code. It stores the converted function code in IRP$W_FUNC and uses the converted function code for all further checking it performs. EXE$QIO stores the function modifiers specified in the FUNC argument in IRP$W_FUNC without change.

16. If the device is not spooled, shareable, or file-oriented, EXE$QIO does not perform any additional privilege checks. Otherwise, it verifies that the process has the necessary privilege to access the device based on whether the I/O function is physical, logical, or virtual.

17. If the request specifies a diagnostic buffer, EXE$QIO allocates the buffer and stores its address in IRP$L_DIAGBUF.

The device-independent preprocessing is complete. EXE$QIO invokes FDT routines to perform device-dependent preprocessing.

23.6.2 FDT Routines

The primary purpose of FDT routines is to validate and process the device-dependent $QIO parameters, P1 to P6. A device driver can include custom FDT routines or use some of the general-purpose routines that are part of the VMS executive. Regardless of the location of FDT routines, they are logically device-dependent extensions of the $QIO system service.

EXE$QIO searches the FDT entries looking for a mask that specifies the function code. When such a mask is found, EXE$QIO invokes the associated FDT routine. If the FDT routine returns control to EXEQIO, EXEQIO continues its search. Successive FDT routines are invoked until an FDT routine invokes one of the routines that terminates FDT processing. These routines are described in the next section.

Note that no FDT entry marks the end of the FDT. It is possible for the search of the FDT to continue past the end of the FDT. Such an occurrence would be an error and would cause unpredictable results.

FDT routines execute in the context of the process that requested the $QIO system service. Therefore, they have access to data in the process's P0 and P1 address space. FDT routines communicate information about the I/O request to the driver through IRP fields. FDT routines can also modify I/O database structures associated with the device assigned to the channel.

FDT routines for direct I/O (I/O done directly between a user buffer and the device) ensure that each buffer page is locked into memory by incrementing its reference count in the page frame number (PFN) database (see Chapter *Memory Management Data Structures*).

In the case of direct I/O, these routines initialize the transfer parameters to describe the buffer as follows:

- IRP$L_SVAPTE contains the system virtual address of the first page table entry that maps the buffer.
- IRP$W_BOFF contains the buffer's offset in bytes from the beginning of that page.

- IRP$L_BCNT is the number of bytes to be transferred.

FDT routines for buffered I/O operations must allocate a buffer from nonpaged pool that will be used by the driver for the actual transfer. If the operation is a buffered write, the FDT routine copies data that is being written to this buffer.

The use of system space buffers permits the device driver to access the data in the buffer when the process is no longer current.

In the case of buffered I/O, these routines initialize the transfer parameters to describe the buffer as follows:

- IRP$L_SVAPTE is the address of the nonpaged pool buffer, which begins with a 12-byte header, shown in Figure 23–6 (see Section 23.7.3.1).
- IRP$W_BOFF is the amount charged against the process's job byte count quota.
- IRP$L_BCNT is the number of bytes to be transferred.

Transfers that can take a long time to complete (such as a terminal read or write) are often implemented as buffered I/O operations, whereas transfers that should complete quickly (such as a disk read or write) are implemented as direct I/O operations. Direct I/O requires locking process pages and page tables into memory, thus tying up the process header, or balance slot, for the duration of the I/O request. Chapter *The Swapper* contains more information on the complexity of swapping a process with direct I/O in progress.

23.6.3 I/O Completion

It is important to distinguish between completion of the $QIO system service request, which signals either that the I/O is underway or that the service was requested incorrectly, and the completion of the I/O request itself.

Passing a status in R0, EXE$QIO returns through the change mode dispatcher to the access mode from which it was requested. If the status is not a success, control returns to the image at a point following its service request. If the status is a success and the image requested the asynchronous form ($QIO) of the service, control returns to the image, which later will request the $SYNCH system service or an event flag wait service to await I/O completion. If the status is a success and the image requested the synchronous form ($QIOW), the executive places the process into an event flag wait until the I/O completes (see Section 23.6.4). Chapter *System Service Dispatching* has more information on how a synchronous system service waits a process.

An I/O request can complete when EXE$QIO returns to its requestor, or it can be passed on to the device driver for device action or some other processing. An FDT routine typically determines how an I/O request completes:

1. If an I/O request is ill formed, the FDT routine invokes EXE$ABORTIO, in module SYSQIOREQ, to abort the request.
2. If no device action or further processing is necessary, or a device-specific error occurs in the FDT routine, it invokes either EXE$FINISHIO or EXE$FINISHIOC, both in module SYSQIOREQ, to complete the I/O request.

3. If an ACP function was requested, or if ACP intervention is required before the I/O request can be delivered to the driver, the FDT routine invokes EXE$QIOACPPKT, in module SYSQIOREQ.
4. If device action or some further driver processing is needed, the FDT routine invokes either EXE$QIODRVPKT or EXE$ALTQUEPKT, both in module SYSQIOREQ.

The next section explains how EXE$QIO itself completes an I/O request. Subsequent sections describe each of the routines that an FDT routine can invoke to complete the request in other ways.

23.6.3.1 $QIO Completion by EXE$QIO

EXE$QIO itself completes an I/O request only if an error occurs.

As discussed previously, EXE$QIO makes certain checks before it allocates an IRP; for example, the CHAN argument must specify a usable channel. If EXE$QIO detects an error before allocating an IRP, it takes the following steps:

1. It invokes SCH$POSTEF, in module POSTEF, to set the event flag specified by the EFN argument.
2. It returns an error status in R0 to the requestor.

If EXE$QIO detects an error after it has allocated an IRP, it aborts the I/O, as described in Section 23.6.3.2.

23.6.3.2 Aborting an I/O Request—EXE$ABORTIO

If EXE$QIO (after it has allocated an IRP) or an FDT routine detects a device-independent error (for example, insufficient privilege), it loads the final status of the system service in R0 and invokes EXE$ABORTIO, in module SYSQIOREQ, to abort the I/O. EXE$ABORTIO takes the following steps:

1. EXE$ABORTIO acquires the device fork lock, raising IPL to fork IPL.
2. It clears IRP$L_IOSB, the address of the IOSB, so that no status is written to it.
3. It clears ACB$V_QUOTA in IRP$B_RMOD and increments the process's AST quota if the bit was set. This prevents a user-specified AST procedure from being called.
4. It inserts the IRP in the per-CPU I/O postprocessing queue and requests an IPL$_IOPOST interrupt (see Section 23.7). During postprocessing, any quotas charged will be restored and buffers deallocated or unlocked, if necessary. Chapter *Software Interrupts* describes the two types of I/O postprocessing queues.
5. EXE$ABORTIO releases the device fork lock.
6. It lowers IPL to 0 and returns to the system service requestor.

The effect of these steps is to complete the system service request without performing any I/O operation. EXE$ABORTIO subjects the IRP to I/O postprocessing because EXE$ABORTIO could have been invoked at any stage of I/O preprocessing. I/O postprocessing guarantees that the I/O request will be aborted cleanly, that is, any allocated buffers will be returned, any locked pages will be released, and so on.

23.6.3.3 Completing the I/O Request in the FDT Routine—EXE$FINISHIO(C)

Some I/O requests can be completed by an FDT routine without the need for driver processing and device operation. There are two circumstances under which this can occur:

- If the FDT routine detects a device-specific error, for example, a buffer not properly aligned
- If the FDT routine can perform all requested operations, for example, an IO$_SENSEMODE operation that returns only fields in the UCB

The FDT routine takes essentially the same action in both cases; the difference is in the status it returns.

The FDT routine invokes either EXE$FINISHIO or EXE$FINISHIOC, both in module SYSQIOREQ. These are alternative entry points to the same routine.

1. EXE$FINISHIOC clears R1. It then continues as if entry had been at EXE$FINISHIO.
2. EXE$FINISHIO increments the operation count in the UCB.
3. It stores R0 and R1 in IRP$L_IOST1 and IRP$L_IOST2. R0 on entry to both routines contains the first longword to be stored in the IOSB. R1 on entry to EXE$FINISHIO contains the second longword to be stored in the IOSB.
4. EXE$FINISHIO acquires the device fork lock, raising IPL to fork IPL.
5. It loads the success status SS$_NORMAL in R0 as the final status of the $QIO system service. Note that the final status of the I/O operation, now in the low-order word of IRP$L_IOST1, can be a failure status.
6. It inserts the IRP in the per-CPU I/O postprocessing queue and requests an IPL$_IOPOST interrupt.
7. EXE$FINISHIO releases the device fork lock.
8. It lowers IPL to 0 and returns to the system service requestor.

23.6.3.4 Entering the Driver's Start I/O Routine—EXE$QIODRVPKT

To initiate I/O on a device, an FDT routine typically invokes EXE$QIODRVPKT. If the device is idle, EXE$QIODRVPKT invokes the driver's start I/O routine; otherwise, EXE$QIODRVPKT inserts the IRP on the wait queue in the device's unit control block (UCB).

EXE$QIODRVPKT invokes EXE$INSIOQ, in module SYSQIOREQ, which performs the following actions:

1. It acquires the device fork lock, raising IPL to fork IPL.
2. If UCB$V_BSY in UCB$L_STS is set, indicating that the device is busy, EXE$INSIOQ invokes EXE$INSERTIRP, in module SYSQIOREQ, to insert the IRP on the device's queue of pending I/O requests. The queue, whose listhead is at UCB$L_IOQFL, is ordered according to the base priority of the process that requested the I/O. When EXE$INSERTIRP returns, control is transferred to step 5.

3. If the device is idle, EXE$INSIOQ marks it busy by setting UCB$V_BSY in UCB$L_STS and invokes IOC$INITIATE, in module IOSUBNPAG, to initiate device I/O.

   IOC$INITIATE performs the following steps:

   a. It saves the IRP address in UCB$L_IRP.

   b. It copies IRPL_SVAPTE, IRPL_BCNT, and IRP$W_BOFF to UCB$L_SVAPTE, UCB$L_BCNT, and UCB$W_BOFF. This step is an optimization for direct I/O operations and is unnecessary for most buffered I/O operations.

   c. It clears UCB$V_TIMOUT and UCB$V_CANCEL in UCB$L_STS. Table 23–2 explains the significance of these and other flags in UCB$L_STS.

   d. If a diagnostic buffer is associated with the current I/O request, IOC$INITIATE obtains its address and records the current system time in it as the operation start time.

      A process requests diagnostic I/O to a driver that supports it by making a physical I/O request and specifying the address of a diagnostic buffer as a function-specific parameter. If the requesting process has the DIAGNOSE privilege, EXE$QIO allocates a system diagnostic buffer from nonpaged pool and records its address in IRP$L_DIAGBUF. EXE$QIO also sets IRP$V_DIAGBUF in IRP$L_STS. Various components of the VMS executive support diagnostic I/O by filling in different parts of the diagnostic buffer if IRP$V_DIAGBUF is set. The system diagnostic buffer is copied to the requesting process's diagnostic buffer by the I/O completion AST routine, as described in Section 23.7.3.2.

   e. IOC$INITIATE gets the address of the driver dispatch table (DDT) from the UCB, locates the driver's start I/O routine through DDT$L_START, and invokes it.

Table 23–2 Flags in UCB$L_STS

Flag	Meaning if Set
UCB$V_BSY	Device start I/O routine is currently processing an I/O request
UCB$V_INT	An interrupt is expected from this device
UCB$V_TIM	This device has an I/O operation being timed
UCB$V_TIMOUT	This device has timed out
UCB$V_CANCEL	Current I/O on the device has been canceled
UCB$V_POWER	The system recovered from a power failure

4. The driver's start I/O routine eventually returns to IOC$INITIATE, and IOC$INITIATE returns to EXE$INSIOQ, as discussed in Chapter *I/O Processing*.

5. EXE$INSIOQ releases the device fork lock, restoring the IPL at entry, and returns control to EXE$QIODRVPKT.

6. EXE$QIODRVPKT restores IPL to 0 and returns to the image that requested the I/O. The status returned in R0 indicates that the I/O request was queued to the driver successfully. The $QIO requestor cannot determine the status of the I/O operation until the VMS I/O postprocessing routine writes the I/O status block for this I/O request.

23.6.3.5 Entering the Driver's Alternate Start I/O Routine—EXE$ALTQUEPKT

An FDT routine invokes EXE$ALTQUEPKT to enter a driver's alternate start I/O routine. The terminal driver's write FDT routine, for example, enters the alternate start I/O routine when a write to a full duplex terminal is requested.

A driver can queue an I/O request to another driver by using that driver's alternate start I/O routine in conjunction with an I/O postprocessing system completion routine (see Section 23.7.1). NETDRIVER, for example, uses this mechanism to interact with its communications drivers.

A driver with an alternate start I/O routine must be able to deal with multiple concurrent I/O requests.

EXE$ALTQUEPKT invokes the driver's alternate start I/O routine regardless of the setting of the UCB$V_BSY bit, as follows:

1. It acquires the device fork lock, raising IPL to fork IPL.
2. EXE$ALTQUEPKT gets the address of the driver's DDT from the UCB, locates the driver's alternate start I/O routine through offset DDT$L_ALTSTART, and invokes it.
3. When the driver's alternate start I/O routine returns, EXE$ALTQUEPKT releases the device fork lock, restoring the IPL at entry, and returns to its invoker, the FDT routine.

23.6.3.6 Initiating ACP I/O—EXE$QIOACPPKT

Some I/O requests must be processed by a device's ACP. For example, when a file system function such as IO$_ACCESS is requested, or when a window turn is required to map a requested virtual block number (VBN) to a logical block number (LBN), a disk driver requires assistance from the XQP, which is a form of ACP (see Chapter *Overview of the I/O Subsystem*). A window turn updates file retrieval information in the WCB (see Section 23.8.2).

An ACP transforms an I/O request into one more suitable for the driver and queues the request to the driver. Alternatively, it requests one or more I/O operations itself to perform the requested function. After it has performed the function, the ACP initiates I/O postprocessing.

EXE$QIOACPPKT is invoked by file system FDT routines in module SYSACPFDT when an I/O request requires action by the XQP. It is also invoked by other FDT routines for I/O requests that require ACP assistance. EXE$QIOACPPKT performs the following actions:

1. It locates the volume control block (VCB) of the device from the UCB. From the VCB, it locates the ACP queue block (AQB) and tests AQB$L_ACPPID. Chapter *Overview of the I/O Subsystem* offers more information on ACPs, the VCB, and the AQB.

EXE$QIOACPPKT makes a distinction between the XQP and other ACPs. The XQP runs in the I/O requestor's process context, whereas other ACPs run as separate processes. In the XQP's AQB, AQB$L_ACPPID is zero. For other ACPs, the field contains the process ID of the ACP.

2. If AQB$L_ACPPID is nonzero, EXE$QIOACPPKT inserts the IRP into the tail of the interlocked I/O request queue at AQB$L_ACPIQ.

   a. If the queue was not empty, EXE$QIOACPPKT returns a successful status to its requestor, indicating that the I/O request is queued.

   b. If this IRP is the first to be inserted into the queue, EXE$QIOACPPKT gets the ACP's PID from AQB$L_ACPPID and invokes SCH$WAKE, in module RSE, to wake up the ACP. If SCH$WAKE returns a success status, EXE$QIOACPPKT returns a success status for the $QIO request.

      Later, when the ACP is placed into execution, the ACP removes the request from its queue, performs the requested function, and initiates I/O postprocessing by queuing the IRP to the systemwide postprocessing queue. Section 23.7 and Chapter *I/O Processing* discuss I/O postprocessing.

3. If AQB$L_ACPPID is zero, EXE$QIOACPPKT enters EXE$QXQPPKT, in module SYSQIOREQ. EXE$QXQPPKT generates a file system request in the context of the current process:

   a. Using the portion of the IRP that begins at offset IRP$L_IOQFL as an ACB, it stores the procedure value of the XQP procedure F11X$RCV_PKT, in module [F11X]DISPAT, in ACB$L_AST.

   b. EXE$QXQPPKT stores the address of the IRP itself in ACB$L_ASTPRM.

   c. It then invokes SCH$QAST, in module ASTDEL, to queue the IRP as a kernel mode AST to the current process.

   d. When SCH$QAST returns, EXE$QXQPPKT lowers IPL to 0 and returns control to the $QIO system service requestor.

Before control returns to the $QIO requestor, the kernel mode AST is delivered and procedure F11X$RCV_PKT is called. F11X$RCV_PKT queues the IRP to the per-process XQP queue in the XQP's data area. If the XQP is not busy servicing another IRP, F11X$RCV_PKT resumes the XQP's thread of execution; otherwise F11X$RCV_PKT simply returns, dismissing the kernel mode AST. When the XQP has finished processing its current I/O request, it removes the next IRP from its per-process queue and begins processing it.

When the XQP has serviced the request, it performs I/O postprocessing by invoking special entry points in IOC$IOPOST, in module IOCIOPOST.

23.6.4 $QIOW System Service

The $QIOW system service is the synchronous form of the $QIO system service. It takes the same arguments as the $QIO system service. EXE$QIOW, the $QIOW system service procedure, executes in its requestor's access mode. It simply copies the arguments on its current stack and requests the $QIO system service.

If the $QIO system service returns an error, the $QIOW system service returns the error status to its requestor. Otherwise, EXE$QIOW invokes EXE$SYNCH_LOOP, in module SYSSYNCH. EXE$SYNCH_LOOP places the process in a wait state until the event flag associated with the I/O request is set. When the process resumes as a result of the setting of the event flag, EXE$SYNCH_LOOP verifies that the IOSB was updated through the completion of the I/O operation. If the IOSB was not updated, EXE$SYNCH_LOOP clears the event flag and places the process in a wait state for the same event flag. The $QIOW request does not complete until the IOSB is updated.

If the IOSB argument was omitted in the $QIOW request, EXE$SYNCH_LOOP returns to the $QIOW requestor once the event flag is set. Note, however, that unless a unique event flag was associated with the $QIOW operation, there is no guarantee that the I/O operation completed; the completion of another operation that was associated with the event flag may have set the event flag.

23.7 I/O Postprocessing

VMS performs I/O postprocessing after an associated driver completes an I/O operation. The I/O postprocessing routine IOC$IOPOST, in module IOCIOPOST, is the interrupt service routine for the IPL$_IOPOST software interrupt. It implements the device-independent steps necessary to complete an I/O request.

Some I/O postprocessing operations, for example, unlocking buffer pages and deallocating buffers, are performed by IOC$IOPOST. Other operations, such as writing the IOSB, are performed by a special kernel mode AST routine, discussed in Section 23.7.3. Chapter *Software Interrupts* gives details on the two types of I/O postprocessing queues. Since the processing for entries in each of the two queues is identical, the following discussion treats them as if they were one.

IOC$IOPOST removes the first IRP in the I/O postprocessing queue. It takes one of two paths, depending upon the value in IRP$L_PID. If the value in IRP$L_PID is negative, IOC$IOPOST performs system I/O completion. If the value in IRP$L_PID is positive, IOC$IOPOST performs normal I/O completion. These two paths are described in the following sections.

23.7.1 System I/O Completion

A negative value in IRP$L_PID is the procedure value of the system completion routine to be invoked when the I/O completes. (System completion routines are also known as end action routines.) IOC$IOPOST invokes this routine. When it returns, IOC$IOPOST removes the next IRP from the queue and processes it.

Various components use system completion routines to perform specialized I/O postprocessing.

23.7.2 Normal I/O Completion

A positive value in IRP$L_PID is the process ID of the I/O requestor. IOC$IOPOST determines the type of I/O operation by testing IRP$V_BUFIO in IRP$W_STS. If the bit is set, the I/O operation is buffered; otherwise, it is direct. IOC$IOPOST performs actions appropriate to the type of I/O operation and then queues a special kernel mode AST to the requestor. The AST routine will perform the completion that must be done in the context of the requestor.

23.7.2.1 Buffered I/O Completion

Buffered I/O involves a transfer to or from a system space buffer in nonpaged pool. IOC$IOPOST takes the following initial steps in the case of buffered I/O:

1. It increments PCB$W_BIOCNT, the number of concurrent buffered I/O requests allowed.
2. If IRP$V_FILACP in IRP$W_STS is set, IOC$IOPOST also increments PCB$W_DIOCNT, the number of concurrent direct I/O requests allowed. Routine BUILDACPBUF, in module SYSACPFDT, sets this bit for most ACP I/O requests. FDT routines for most common ACP function requests invoke BUILDACPBUF.
3. IOC$IOPOST invokes EXE$CREDIT_BYTCNT, in module EXSUBROUT, to restore the byte count quota that was charged for the system buffer. Note that IRP$W_BOFF does not contain a buffer offset in this case; it contains a byte count. The FDT routine that allocated the system buffer stored the size of the buffer in IRP$W_BOFF and charged the job information block (JIB) for the buffer.
4. IOC$IOPOST stores the procedure value of the special kernel mode AST routine in the IRP at offset ACB$L_KAST. The IRP will also be used as an ACB. ACB$L_KAST and IRP$L_WIND are the same offset. At this point, the WCB address is no longer needed and that location can be reused safely.

   The special kernel mode AST routine, in module IOCIOPOST, has two entry points: BUFPOST, for buffered read completion, and DIRPOST, for all others. The first case differs from the others in that data must be copied from the system buffer to the process buffer before the process is informed that the I/O is complete. In the case of a buffered write, there is no need to copy data between the process buffer and the system buffer. It was copied earlier from the process buffer to the system buffer by an FDT routine. In the case of direct I/O, there is no system buffer.

   It is possible that there was no need for a system buffer; an I/O request with no transfer of data is usually performed as a buffered I/O request. If a buffer was needed, its address is in IRP$L_SVAPTE.

   — If IRP$L_SVAPTE is nonzero and IRP$V_FUNC in IRP$W_STS is set, the I/O function is a read requiring a buffer. In this case, IOC$IOPOST stores the procedure value of BUFPOST in ACB$L_KAST.

   — Otherwise, IOC$IOPOST stores the procedure value of DIRPOST in ACB$L_KAST. If IRP$L_SVAPTE is nonzero, IOC$IOPOST deallocates the buffer.
5. It performs the steps described in Section 23.7.2.3.

23.7.2.2 Direct I/O Completion

Direct I/O requests involve the transfer of data directly to or from the process buffer, which can be paged. Since paging must not occur during the processing of the I/O request, the pages are locked in memory by one of the FDT routines invoked by EXE$QIO.

For direct I/O operations other than swapping and paging I/O, which are discussed in Chapters *Paging Dynamics* and *The Swapper*, IOC$IOPOST takes the following initial steps:

1. It performs the steps necessary to handle segmented transfers, if needed, as described in Section 23.8.
2. It determines the number of pages the direct I/O buffer occupies from IRP$W_BOFF and IRP$L_BCNT. IRP$L_SVAPTE contains the address of the first page table entry that maps the buffer. It unlocks the pages by invoking MMG$UNLOCK, in module IOLOCK, which decrements the pages' associated reference counts in the PFN database (see Chapter *Memory Management Data Structures*). This step can result in the pages being placed on the free or modified page list.
3. An IRP by itself can only describe one direct I/O buffer. If a direct I/O request has more than one buffer, an FDT routine allocates one or more IRP extensions (IRPEs) to describe them. Each IRPE can describe two buffers. An IRP with an IRPE has bit IRP$V_EXTEND set in IRP$W_STS and the address of the IRPE in IRP$L_EXTEND. Similarly, each IRPE can point to another IRPE.

   IOC$IOPOST tests whether IRPEs are present; if so, it unlocks the additional buffers they describe.

   No drivers that are part of [*censored*] VMS use IRPEs.
4. It increments PCB$W_DIOCNT, the number of allowed concurrent direct I/O requests.
5. It stores the procedure value of DIRPOST in ACB$L_KAST.
6. It performs the steps described in Section 23.7.2.3.

23.7.2.3 Final Steps in IOC$IOPOST

IOC$IOPOST performs the same final steps for each buffered and direct I/O request:

1. If appropriate, it invokes SCH$POSTEF, in module POSTEF, to set the specified event flag for the process whose I/O just completed.
2. It queues a postprocessing special kernel mode AST to the process.

Whether IOC$IOPOST or its AST routine sets the event flag is determined by the type of flag: if the flag is local, IOC$IOPOST sets it; otherwise, the AST routine sets it.

A potential synchronization problem could occur if a process whose event flag wait is satisfied executes before the postprocessing AST routine copies possible buffered input to a process buffer and records status in the IOSB. This race condition could occur if multiple processes were waiting for a common event flag associated with an I/O request and one of them executed before the process that

requested the I/O could execute the postprocessing AST routine. IOC$IOPOST avoids this race condition by not setting a common event flag itself; instead, its AST routine does.

IOC$IOPOST sets ACB$V_KAST in IRP$B_RMOD to indicate that this is a special kernel mode AST and invokes SCH$QAST, in module ASTDEL, to queue the AST to the process identified by the IRP$L_PID field. The IRP is used as the ACB for SCH$QAST, as described in Chapter *ASTs*. Except for ACB$L_KAST and ACB$V_KAST, IOC$IOPOST does not change any fields in the IRP/ACB.

IOC$IOPOST attempts to remove another IRP from the I/O postprocessing queue. If it is successful, it processes that IRP. Otherwise, it executes a `CALL_PAL REI` instruction to exit the interrupt service routine.

23.7.3 I/O Completion Special Kernel Mode AST Routine

The I/O completion special kernel mode AST routine has two entry points: BUFPOST and DIRPOST. BUFPOST performs certain steps unique to buffered read completion and then falls into DIRPOST.

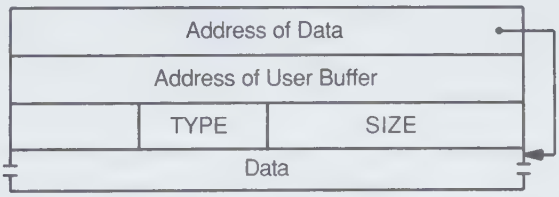
23.7.3.1 Buffered Read Completion

BUFPOST copies data from system buffers allocated by an FDT routine to user buffers in per-process address space. BUFPOST processes three types of system buffer, identified by IRP$W_STS bits:

- Simple buffer—indicated by IRP$V_COMPLEX clear. Simple buffers are used for most buffered and diagnostic I/O operations.
- Complex buffer—indicated by IRP$V_COMPLEX set and IRP$V_CHAINED clear. Most ACP FDT routines in module SYSACPFDT use complex buffers to process ACP I/O requests.
- Chained complex buffer—indicated by IRP$V_COMPLEX and IRP$V_CHAINED set. Chained complex buffers are used by network device drivers.

When a simple buffer is associated with an I/O request, IRP$L_SVAPTE contains its address and IRP$L_BCNT contains the number of bytes of data in the buffer. Figure 23–6 shows the layout of a simple buffer.

Figure 23–6 Layout of a Simple Buffer



The first longword of the buffer points to the data, beyond the header. The second longword contains the address of the user buffer. The next word contains the size of the simple I/O buffer. The next byte contains the type, typically DYN$C_BUFIO. The next byte is spare. The rest of the buffer contains the data.

BUFPOST invokes routine MOVBUF, in module IOCIOPOST, to move the data. MOVBUF takes the following steps:

1. It verifies that the user buffer is still write-accessible to the access mode in IRP$B_RMOD.

   If it is not write-accessible, MOVBUF modifies the final status in IRP$L_IOST1 to be SS$_ACCVIO.

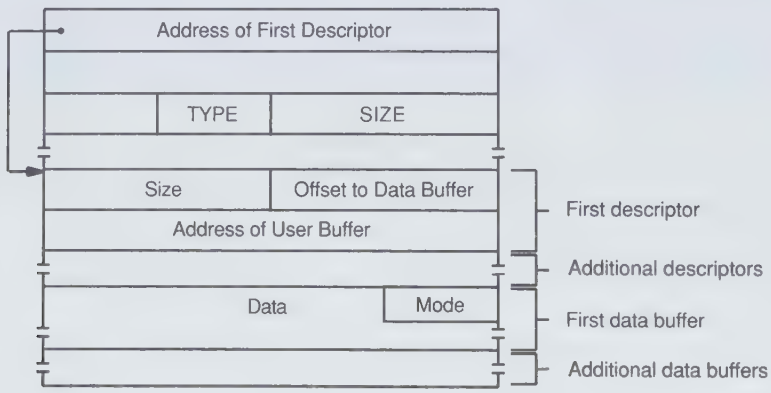
2. Otherwise, MOVBUF copies the data from the system buffer to the user buffer.
3. It deallocates the system buffer to nonpaged pool.
4. It returns to its invoker.

If the I/O request is a mailbox read (IRP$V_MBXIO in IRP$W_STS is set), BUFPOST invokes SCH$RAVAIL, in module MUTEX, to declare the mailbox resource available in case a process is waiting for this resource. Resources are discussed in Chapter *Scheduling*.

When a complex buffer is associated with an I/O request, IRP$L_SVAPTE contains its address and IRP$L_BCNT contains the number of descriptors in the packet.

The layout of a complex buffer is shown in Figure 23–7. The first longword points to the first descriptor. The second longword is ignored by BUFPOST. The third longword contains the size and type. There can be space between the third longword and the first descriptor. The rest of the buffer consists of descriptors and the associated data buffers.

Figure 23–7 Layout of a Complex Buffer



Each descriptor has the same format. The offset field contains the offset from the start of the descriptor to the data buffer in the packet. The size field contains the number of bytes in the data buffer; the size can be zero. The user buffer address is the address of the per-process space user buffer. The first byte in the data buffer is the access mode associated with the user buffer.

One common instance of the complex buffer is the ACP I/O buffer (AIB) used by the XQP. In the AIB, the third longword is followed by an access rights block (ARB) copied from the requestor's PCB. The descriptors apply to input data as well as output data. In the case of input data, the size field in the descriptor is set to zero before the IRP is completed by the file system. The file system can also reduce the count of descriptors in IRP$L_BCNT; this is done when the last descriptors are for input data. Since the size contained in the third longword of the buffer reflects the entire buffer, no space is lost when the buffer is deallocated to nonpaged pool.

BUFPOST processes the buffer in the following steps:

1. It gets the address of the first descriptor.
2. If the size field is zero, BUFPOST goes to step 5.
3. If the user buffer is write-accessible, BUFPOST transfers the data from the data buffer to the user buffer.
4. If the user buffer is not write-accessible, BUFPOST modifies the final status in IRP$L_IOST1 to be SS$_ACCVIO and goes to step 6.
5. If there are more descriptors, BUFPOST gets the address of the next descriptor and then goes to step 2.
6. It deallocates the buffer to nonpaged pool.

When a chained complex buffer is associated with an I/O request, IRP$L_SVAPTE contains the address of the first chained complex buffer and IRP$L_BCNT contains the size of the user buffer.

Chained complex buffers are used by some of the communications drivers. They provide a mechanism for one logical buffer to be split into several segments that are not combined until they are transferred to the user buffer.

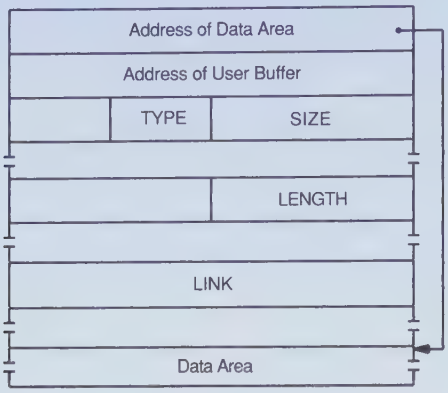
The layout of a chained complex buffer is shown in Figure 23–8. The first longword contains the address of the data area. The second longword contains the address of the user buffer; this field is valid only in the first descriptor in the chain. CXB$W_SIZE contains the size of the chained complex buffer. CXB$B_TYPE contains the type, DYN$C_CXB. CXB$W_LENGTH contains the size of the data area. CXB$L_LINK contains the address of the next chained complex buffer in the chain; zero indicates the end of the chain.

BUFPOST processes the chained complex buffers in the following manner:

1. It verifies that the user buffer is write-accessible to the access mode in IRP$B_RMOD.
2. If the user buffer is not write-accessible, BUFPOST modifies the final status in IRP$L_IOST1 to be SS$_ACCVIO. It then goes to step 6.
3. If the user buffer is write-accessible, BUFPOST sets CXB$W_LENGTH to be the smaller of the amount of space left in the user buffer and the original contents of CXB$W_LENGTH.
4. It moves that amount of data from the data area to the user buffer and reduces the amount of space left in the user buffer by the amount transferred.

5. If there is space left in the user buffer, BUFPOST moves to the next buffer. If there is a next buffer, BUFPOST goes to step 3.
6. BUFPOST deallocates all the buffers to nonpaged pool.

Figure 23–8 Layout of a Chained Complex Buffer



23.7.3.2 Common Completion

DIRPOST performs the completion common to buffered and direct I/O requests:

1. It increments either PHD$L_DIOCNT or PHD$L_BIOCNT, the process's cumulative totals of completed direct I/O and buffered I/O requests.
2. If a user's diagnostic buffer was associated with the I/O request, DIRPOST invokes routine MOVBUF to copy the diagnostic information from the system diagnostic buffer to the user's diagnostic buffer. DIRPOST then deallocates the system diagnostic buffer. The system diagnostic buffer has the same format as a simple buffered I/O buffer.
3. It decrements the CCB count of I/O requests in progress on this channel.
4. If this was the last I/O for the channel and there is a deaccess request for the channel pending, DIRPOST queues that deaccess request to the ACP by invoking IOC$WAKACP, in module IOCIOPOST.
5. If the I/O request specified an IOSB, DIRPOST copies the quadword at IRP$L_IOST1 to the IOSB.
6. If a common event flag is associated with the I/O request, DIRPOST invokes SCH$POSTEF to set the flag.
7. If any IRPEs were used, it deallocates them.
8. If ACB$V_QUOTA is set in IRP$B_RMOD, then the user requested AST notification of I/O completion. The AST procedure value and the optional AST argument were originally stored in the IRP (now used as an ACB). DIRPOST invokes SCH$QAST to queue the IRP as an ACB, this time for a normal AST in the access mode at which the I/O request was made.
9. Otherwise, if ACB$V_QUOTA is clear, DIRPOST deallocates the IRP/ACB to nonpaged pool.

10. It returns to its invoker, SCH$ASTDEL in module ASTDEL.

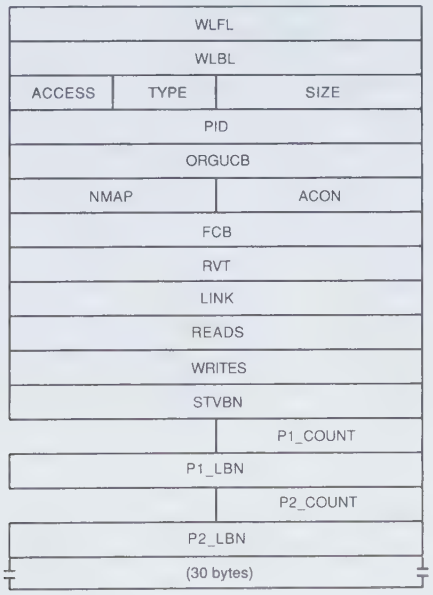
23.8 Segmented Virtual and Logical I/O

Under certain circumstances, the I/O subsystem must break I/O transfer requests involving a block-addressable mass storage device into segments and pass the request to the device driver segment by segment. This section describes the means by which such requests are segmented and successive segments are passed on to a device driver.

A file is stored on such a device in a series of blocks. There are three ways of referring to the blocks: file-relative (virtual), volume-relative (logical), and absolute (physical). An image performing I/O to a file describes its request in terms of the starting VBN and the number of bytes to be transferred. The I/O subsystem must convert the VBN into its corresponding LBN for the device driver. For some devices, the LBN must be converted into the corresponding physical block number.

A logically contiguous series of blocks in a file is called an extent. An extent is described by its starting LBN and the number of blocks in it. Most files are made up of multiple extents; a physically contiguous file has only one extent. Each file has an on-disk data structure called a file header that lists the extents that make up the file. When a file is opened, information about its extents is copied from the file header into the WCB, shown in Figure 23–9. If the image's I/O request crosses a file extent boundary, the I/O subsystem must break the request into segments, each of which fits within one extent.

Figure 23–9 Window Control Block



Certain mass storage devices and their associated drivers cannot handle transfers greater than 64K bytes at one time. In this case the I/O subsystem must break the transfers into segments no greater than 64K. Note that the request can already have been segmented to fit within file extents, which may be greater than 64K bytes.

23.8.1 Segmentation by FDT Routines

Usually, a mass storage device driver specifies the following FDT routines: ACP$READBLK for reads and ACP$WRITEBLK for writes, both in module SYSACPFDT. These routines store the total byte count of the request in the original byte count field of the IRP, IRP$L_OBCNT, and clear the accumulated byte count field of the IRP, IRP$L_ABCNT.

23.8.1.1 Segmenting Virtual I/O

If the transfer is a virtual I/O transfer, these routines then invoke IOC$MAPVBLK, in module IOSUBRAMS, to perform the actual conversion from VBNs to LBNs. IOC$MAPVBLK (see Section 23.8.2) returns the number of bytes not mapped.

If the number of bytes mapped is nonzero, each FDT routine takes the following steps:

1. It computes the number of bytes mapped by subtracting the number of bytes not mapped from IRP$L_OBCNT and stores this number in IRP$L_BCNT.
2. It stores the starting LBN in IRP$L_IOST1.
3. It stores the starting VBN in IRP$L_SEGVBN.
4. It converts the I/O function to the equivalent physical I/O function.
5. It takes the steps discussed in Section 23.8.1.2.

If the number of bytes mapped by IOC$MAPVBLK is zero, the FDT routines store the starting VBN in IRP$L_SEGVBN and the number of bytes not mapped (in this case, the total number of bytes requested) in IRP$L_BCNT and then invoke EXE$QIOACPPKT, in module SYSQIOREQ, to send the IRP to the ACP.

When the file system processes this IRP, it detects that the WCB does not map the requested virtual range and performs a window turn. It reads the file header to obtain the mapping information necessary for the transfer in question and stores the information in the WCB, replacing other mapping information already contained there. The file system performs the equivalent steps that IOC$MAPVBLK performs and then queues the IRP to the driver. Note that the number of bytes mapped at this point is nonzero.

23.8.1.2 Segmenting Logical and Physical I/O

If the function is not a physical I/O function, the FDT routines convert it to the equivalent physical I/O function. The FDT routines then take the steps necessary to handle transfers greater than 64K bytes, as discussed in Section 23.8.3. Note that these steps are not required for all disk devices.

The routines then queue the IRP to the driver. The driver performs the transfer without regard for whether the entire range is to be transferred. IOC$IOPOST will check whether the entire range has been transferred when the driver completes the I/O request and will take the necessary action, as described in Section 23.8.4.

23.8.2 IOC$MAPVBLK

IOC$MAPVBLK uses the information passed (via registers and the IRP) to convert the VBNs to LBNs. The goal is to convert the starting VBN to the related LBN. The gating factor is the information stored in the WCB (the address of the WCB is obtained from CCB$L_WIND) that was created by the file system when the file was opened.

If the WCB contains enough mapping information to convert the entire virtual range of the transfer into corresponding LBNs on the volume, then the virtual I/O transfer will be handled directly by the driver and IOC$IOPOST, even if the transfer consists of several logically noncontiguous pieces. If the WCB does not contain enough information to completely map the virtual range of the transfer, the intervention of the file system will be required at some time to complete the transfer. This intervention is known as a window turn.

Because a deadlock situation could occur if a file mapped by the memory management subsystem required a window turn, the memory management subsystem must avoid window turns. To do this, each file mapped by the memory management subsystem must have all its mapping information in the WCB. A special, large variation of the WCB is used, called a cathedral window.

IOC$MAPVBLK can encounter five possible cases:

- The virtual range is logically contiguous and the WCB contains the needed mapping information. In this case, all that IOC$MAPVBLK needs to do is convert the starting VBN into the related LBN. The driver can transfer the data without further conversion of VBNs into LBNs.
- The WCB contains mapping information for the beginning of the virtual range, but more than two map entries are required to map the range. In this case, IOC$MAPVBLK converts the starting VBN into the related LBN. The driver can transfer the start of the virtual range but will need further conversion of VBNs into LBNs to transfer the rest of the range.

  IOC$MAPVBLK uses only the map entry that maps the starting VBN and the next map entry, if that map entry is logically contiguous with its predecessor. Since the block count field in the map entry is a word in size, it is possible that a logically contiguous range will require more than one map entry to cover the entire logical range.
- The WCB contains mapping information for the beginning of the virtual range but not for the entire virtual range. In this case, IOC$MAPVBLK converts the starting VBN into the related LBN. The driver can transfer the start of the virtual range but will need further conversion of VBNs into LBNs to transfer the rest of the range.

In this case, the virtual range may be logically contiguous, but not enough mapping information is contained in the WCB to verify this. A window turn will be needed later.

- The virtual range is not logically contiguous, but the WCB does contain mapping information for the beginning of the virtual range. IOC$MAPVBLK handles this case in the same way it handles the previous case.

  The driver can transfer the start of the virtual range but will need further conversion of VBNs into LBNs to transfer the rest of the range. The WCB may or may not contain the needed information. If it does not, a window turn will be needed. Whether a window turn will be needed later is irrelevant at this point.

- The mapping information that maps the first virtual block in the range to its logical counterpart is not in the WCB. A window turn is needed before any data can be transferred.

In all five cases, IOC$MAPVBLK returns the number of bytes not mapped. If the number of bytes mapped is nonzero, IOC$MAPVBLK also returns the starting LBN.

23.8.3 Segmenting Transfers Greater Than 64K Bytes

VMS supports I/O transfers greater than 64K bytes for mass storage devices, even though a device and its driver may only support transfers up to 64K bytes. This is done by breaking the transfer into segments no larger than the maximum transfer size supported by the driver. The UCB$L_MAXBCNT field contains the largest transfer size supported by the driver. If it is zero, it is assumed to be 65,024 (64K bytes minus 512).

If the IRP$L_BCNT field is greater than the maximum transfer size specified by UCB$L_MAXBCNT, the FDT routines set IRP$L_BCNT to the maximum transfer size accepted by the driver. Otherwise, they do not modify IRP$L_BCNT. Remember that the FDT routines store the requested size in IRP$L_OBCNT, as noted in Section 23.8.1.

As a result, the first transfer will be the size specified by UCB$L_MAXBCNT. The remainder will be transferred as a result of the steps taken by IOC$IOPOST, as described in Section 23.8.4.

23.8.4 IOC$IOPOST Processing of Segmented Transfers

Whenever IOC$IOPOST encounters an IRP for a direct I/O data transfer request, it determines if another segment must be transferred by comparing the original byte count to the number of bytes transferred thus far. If the difference is not zero, another segment must be transferred. If the two numbers agree, the request is completed exactly like other direct I/O requests.

If the two numbers do not agree, IOC$IOPOST prepares the IRP for the transfer of the next segment by taking the following steps:

1. If the transfer is a virtual I/O transfer, IOC$IOPOST invokes IOC$MAPVBLK.

The same five cases exist here as when IOC$MAPVBLK is invoked by the FDT routines. IOC$IOPOST takes the equivalent steps in each case for the transfer that starts at the VBN in IRP$L_SEGVBN. If there is a total mapping failure of the remaining transfer, IOC$IOPOST invokes IOC$QTOACP to pass the IRP to the ACP. Otherwise, IOC$IOPOST continues.

2. It places the lesser of the remaining byte count and the maximum transfer size accepted by the driver in IRP$L_BCNT.
3. It updates the starting VBN in IRP$L_SEGVBN by the number of blocks transferred in the last transfer.
4. It invokes EXE$INSIOQC, in module SYSQIOREQ, to queue the IRP to the driver.

Thus, in a fashion transparent to the requestor, the original request is segmented to satisfy the limitations of the WCB or the maximum transfer size permitted by the device.

23.9 $CANCEL System Service

The $CANCEL system service cancels pending I/O requests on a specified channel. These include queued I/O requests as well as the request in progress. The $CANCEL system service can be requested by an image. It is also requested by the $DASSGN system service, which is requested during image and process rundown. The $CANCEL system service has only the CHAN argument, which specifies the I/O channel on which I/O is to be canceled.

The $CANCEL system service procedure, EXE$CANCEL in module SYSCANCEL, executes in kernel mode. Kernel mode code can request a second form of the $CANCEL system service by calling the system service procedure directly at an alternative entry point, EXE$CANCELN. This form of the system service has two arguments:

- The CHAN argument
- The optional CODE argument, the reason for the cancellation

EXE$CANCELN determines if the CODE argument is present. If it is present, the procedure saves it for later use. Otherwise, the procedure saves a reason code of CAN$C_CANCEL. EXE$CANCEL, on the other hand, always saves a reason code of CAN$C_CANCEL. Once the reason code has been saved, EXE$CANCEL and EXE$CANCELN converge.

1. EXE$CANCEL invokes IOC$VERIFYCHAN, as discussed in Section 23.5.3, to verify the channel.
2. It raises IPL to 2 to block process deletion.
3. It page faults the CCB into memory and acquires the device fork lock, raising IPL to fork IPL and effectively locking the CCB into memory.
4. It searches the IRPs queued to the UCB (starting at UCB$L_IOQFL), looking for those that meet the following criteria:
   — The requesting process ID (PCB$L_PID) matches the process ID in IRP$L_PID.

— The channel number in IRP$W_CHAN matches the requested channel.

— The request is not a virtual request (IRP$V_VIRTUAL in IRP$W_STS is clear). In general, I/O cannot be canceled on disk or tape devices. Drivers for these devices ensure that IRP$V_VIRTUAL is set on all requests that cannot be canceled.

For each IRP that satisfies these criteria, EXE$CANCEL takes the following steps and then resumes the search:

a. It removes the IRP from the queue.

b. It clears the buffered read bit (IRP$V_FUNC in IRP$W_STS) for buffered I/O functions. Since this I/O operation has not been started, there is no data to be transferred to the user's buffers.

c. It places the error status SS$_CANCEL in the low-order word of IRP$L_IOST1 and clears the high-order word. This is the final status of the I/O operation.

d. It inserts the IRP at the tail of the systemwide I/O postprocessing queue and requests an IPL$_IOPOST interrupt (see Section 23.7).

5. After scanning the IRP queue, EXE$CANCEL invokes the driver's cancel I/O routine, whose procedure value is stored in the driver dispatch table. The driver is passed the cancel reason saved at the start of EXE$CANCEL or EXE$CANCELN. The driver should perform any actions appropriate to canceling I/O.

6. EXE$CANCEL tests the device type to determine whether canceling its active request is appropriate. If the device is a disk, it is likely that the request will complete quickly enough that canceling it is unnecessary. If canceling the active request is not appropriate, EXE$CANCEL exits, as described in step 7. Otherwise, EXE$CANCEL continues with step 8.

7. EXE$CANCEL releases the device fork lock. EXE$CANCEL lowers IPL to 0 and returns the success status SS$_NORMAL to its requestor.

8. If there is no outstanding I/O (CCB$W_IOC is zero) and there is no file activity (CCB$L_WIND is zero), EXE$CANCEL exits, as described in step 7. (If there is file activity, then CCB$L_WIND contains the address of the WCB associated with the channel or a process section index. At this point, the distinction is not significant.)

9. If the device is not mounted or is mounted foreign, EXE$CANCEL exits, as described in step 7.

10. If there is a process section associated with the channel, EXE$CANCEL exits, as described in step 7.

11. At this point, EXE$CANCEL has determined that there is a file open on this channel. If WCB$V_NOTFCP in WCB$B_ACCESS is set, EXE$CANCEL exits, as described in step 7.

The WCB$V_NOTFCP bit identifies a WCB created by special routines that run only during system startup. These routines open files before the Files-11 XQP is available. When these files are opened again after the XQP is available, new WCBs are created. The original WCBs are not destroyed and are not used by the XQP.

12. At this point, EXE$CANCEL has determined that there is a user file open on the channel. It attempts to allocate an IRP to request an IO$_ACPCONTROL function. If it cannot allocate an IRP, it does one of two things:
    - If the process does not have resource wait mode enabled, EXE$CANCEL exits, as described in step 7, with a status indicating the reason that EXE$CANCEL could not allocate an IRP.
    - Otherwise, EXE$CANCEL calls SCH$RESOURCE_WAIT_PS to place the process into an RSN$_NPDYNMEM wait, after relinquishing the device fork lock. When the wait completes, EXE$CANCEL returns to step 2.
13. It initializes the IRP as follows:
    a. The process ID of the requestor is set to the value in PCB$L_PID.
    b. The AST procedure value and parameter are cleared (no user AST).
    c. The WCB address is set to the value in CCB$L_WIND.
    d. The UCB address is stored in IRP$L_UCB.
    e. The function code is set to IO$_ACPCONTROL.
    f. The event flag is set to EXE$C_SYSEFN.
    g. The priority is set to the process's base priority.
    h. The IOSB address is set to zero.
    i. The channel number is stored in IRP$W_CHAN.
    j. The I/O is marked as buffered I/O with no buffer.
    k. The access rights block address is set to the value in PCB$L_ARB.

    The I/O function code IO$_ACPCONTROL is special because it has no associated I/O buffer. It is ignored by disk ACPs and the XQP. It is recognized by the magnetic tape ACP as a special I/O abort function (equivalent to invoking the driver's cancel I/O routine) that causes the ACP to abort the mounting of a multivolume tape file.
14. EXE$CANCEL charges the user's buffered I/O quota, PCB$W_BIOCNT, for an I/O request.
15. The device fork lock is released.
16. EXE$CANCEL invokes EXE$QIOACPPKT to queue the packet to the file system. EXE$QIOACPPKT returns control to the requestor of the system service.

23.10 Revision Information

Source Module	Build Date
[SYS]SYSDEVALC.MAR	12-June-1991
[SYS]SYSDASSGN.MAR	12-June-1991
[SYS]SYSASSIGN.MAR	12-June-1991
[SYS]SYSQIOREQ.MAR	12-June-1991
[SYS]IOCIOPOST.MAR	12-June-1991
[SYS]IOSUBRAMS.MAR	12-June-1991
[SYS]SYSCANCEL.MAR	12-June-1991
[SYS]IOSUBPAGD.MAR	12-June-1991
[SYS]EXSUBROUT.MAR	12-June-1991

Chapter 24

I/O Processing

"Open the pod-bay doors, HAL."

Arthur C. Clarke, *2001: A Space Odyssey*

Once a user's I/O request is preprocessed and validated by the Queue I/O Request ($QIO) system service and a device driver's function decision table (FDT) action routine, the VMS executive invokes the driver's start I/O routine to perform the requested function. Chapter *I/O System Services* describes the preprocessing and validation of the I/O request. This chapter describes how a driver's start I/O routine interacts with VMS to perform a user-requested function and how VMS dispatches I/O device interrupts.

24.1 Device Driver Models in VMS

There are two basic device driver models in VMS: the traditional model, in which the entire driver is contained in one image, and the class/port model, in which the driver is implemented by more than one image.

The class/port driver model evolved from the traditional model and retains the same user interface. In the class/port driver model, the class driver performs functions common to a class of device, such as a small computer systems interface (SCSI) disk or a SCSI tape, for example. The port driver contains controller-specific subroutines for the class driver. Chapter *Miscellaneous I/O Topics* gives more information on class/port drivers.

VAX/VMS support routines and VAX MACRO macros for traditional drivers have been ported to [*censored*] VMS to allow Digital to port existing VAX/VMS drivers with minimal change to [*censored*] VMS Version 1.0. Use of this mechanism is reserved to Digital; any other use is unsupported. Moreover, like other internal interfaces, these driver porting mechanisms are subject to change in future releases.

This chapter describes I/O processing in terms of the ported VAX/VMS mechanisms for traditional drivers.

24.2 The Start I/O Routine

The heart of a driver, regardless of driver model, is its start I/O routine. The start I/O routine services I/O requests by interacting with the device controller.

A start I/O routine usually performs at least the following steps:

1. It allocates controller- and device-specific resources.
2. It initiates device activity.
3. It awaits device activity completion.
4. It deallocates controller- and device-specific resources, if allocated in step 1.
5. It initiates I/O request completion processing.

A simple traditional driver's start I/O routine is shown in Example 24–1. How it interacts with the VMS executive to perform some of the preceding steps is the subject of discussion in this chapter. Chapter *Miscellaneous I/O Topics* describes allocation and deallocation of controller- and device-specific resources.

Example 24–1 Simple Start I/O Routine

```
STARTIO:  .JSB_ENTRY <R2,R3,R4,R5>
          .
          .
; Initiate device activity by informing controller
; of required action
          .
          .
          WFIKPCH   DEVTMO,#6     ;Wait for interrupt or timeout
          .                       ;Execution resumes here upon
          .                       ; interrupt
          IOFORK                  ;Request to defer further
          .                       ; processing to a lower IPL
          .
          REQCOM                  ;Initiate I/O request completion
                                  ; processing
```

Example 24–1 shows how the macros WFIKPCH, IOFORK, and REQCOM create the framework for a simple start I/O routine. These and other related macros are documented in *VMS Device Support Manual*. The macros, provided as a convenience to the driver writer, invoke VMS routines described in this chapter.

24.2.1 Driver-Controller Interface

The design of the driver-controller interface determines the actions of a driver's start I/O routine. Chapter *I/O Architecture* describes two categories of driver-controller interface: a simple interface, in which a driver communicates with a controller exclusively through interface registers, and a more sophisticated interface, in which a driver uses command and response buffers in memory in addition to controller interface registers.

The driver used as an example in this section implements a simple driver-controller interface.

24.2.2 Device Activity Initiation

A start I/O routine that implements a simple driver-controller interface initializes one or more interface registers to inform the device controller of required action and to initiate device activity.

A start I/O routine that implements a more complex driver-controller interface might create one or more command messages in memory and access a controller interface register to notify the controller of awaiting command messages.

The VMS executive invokes a traditional driver's start I/O routine when a user requests an I/O operation and the device is idle. Figure 24–1 shows the flow of control from a user image to the start I/O routine and back. Note that in Figures 24–1, 24–2, and 24–3 portions of the start I/O routine that are not relevant to the control flow, but aid in understanding it, are shaded.

The numbers in Figure 24–1 correspond to the numbered steps that follow:

❶ The user image requests the $QIO system service.

❷ The $QIO system service procedure, EXE$QIO in module SYSQIOREQ, validates the function-independent parameters of the request, allocates and builds an IRP, and invokes the driver's FDT routine for the requested function.

❸ The FDT routine validates function-dependent parameters of the I/O request, sets up any necessary I/O buffers, and invokes EXE$QIODRVPKT, in module SYSQIOREQ. Chapter *I/O System Services* provides details on FDT routine processing and the control flow into the driver's start I/O routine.

❹ EXE$QIODRVPKT invokes EXE$INSIOQ, also in module SYSQIOREQ. Chapter *I/O System Services* describes EXE$INSIOQ.

❺ EXE$INSIOQ acquires the fork lock for the UCB and tests UCB$V_BSY. If the device is not busy, it invokes IOC$INITIATE.

❻ IOC$INITIATE locates the driver's start I/O routine and invokes it.

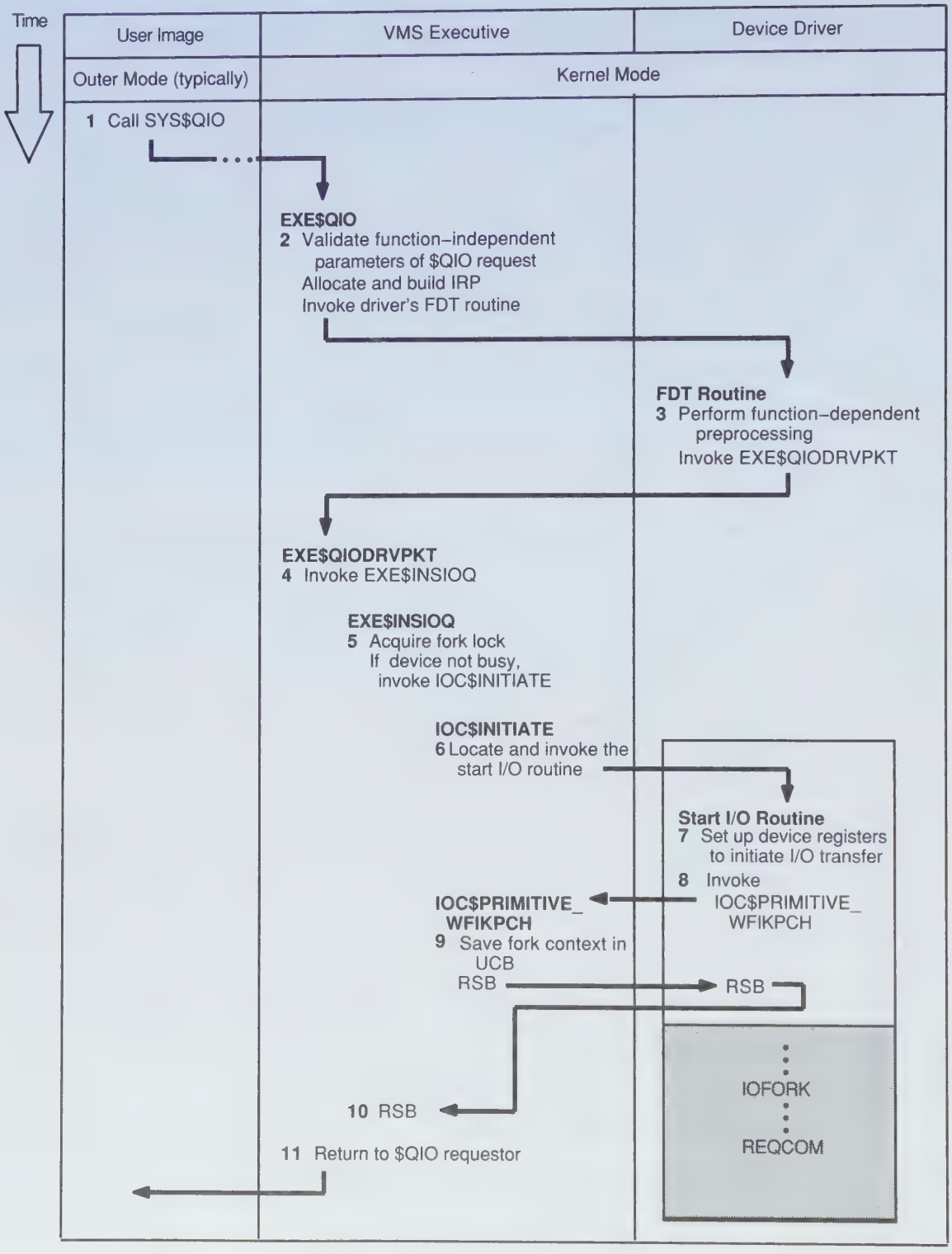
❼ The start I/O routine acquires the device lock to synchronize its activity with device interrupts. It then initializes one or more device interface registers with information such as function requested and address of I/O buffer.

❽ Typically, a simple driver-controller interface includes a bit, sometimes called the GO bit, that indicates to the device that the driver has provided the device with all necessary information for a transfer. The setting of this bit is a cue that the device can start the transfer. After the start I/O routine sets this bit, it must stall until the device signals completion through an interrupt. It does this by invoking the WFIKPCH macro.

The WFIKPCH macro has two required arguments:

- The address of the timeout routine
- The number of seconds within which the interrupt must occur

Figure 24–1 Entering the Start I/O Routine



If the interrupt occurs within the specified time, VMS resumes the start I/O routine at the VAX MACRO statement following the WFIKPCH macro; otherwise, VMS resumes the start I/O routine at the timeout routine specified. The WFIKPCH macro invocation in Example 24–1 expands to the code shown in Example 24–2.

Example 24–2 Expansion of the WFIKPCH Macro

```
;Expansion of WFIKPCH DEVTMO,#6

    MOVL    #6,R1                   ;Timeout time = 6 seconds
    MOVL    (SP)+,R2                ;IPL to restore
    MOVAB   L1,UCB$L_FPC(R5)        ;Processing will resume at
                                    ; L1 after interrupt
    JSB     IOC$PRIMITIVE_WFIKPCH   ;
    RSB                             ;Return to start I/O
                                    ; routine's invoker
                                    ;
L1: .JSB_ENTRY INPUT=<R3,R4,R5>,SCRATCH=<R0,R1,R2>
                                    ;
    BITW    #UCB$M_TIMOUT, -        ;Did interrupt arrive
            UCB$W_STS(R5)           ; on time?
    BNEQ    DEVTMO                  ;If not zero, device
                                    ; timed out
```

Note that when IOC$PRIMITIVE_WFIKPCH is invoked, UCB$L_FPC contains the procedure value of entry point L1, which the WFIKPCH macro generated.

24.2.3 Waiting for Device Activity Completion

A controller that implements a simple driver-controller interface signals completion of each I/O request by generating an I/O interrupt. A controller that implements a more sophisticated interface can signal the completion of multiple I/O requests by creating multiple response messages in memory and generating a single interrupt; each response message holds the completion status of a command message.

This section describes how IOC$PRIMITIVE_WFIKPCH, in module IOSUBNPAG, stalls the simple driver's start I/O routine of Example 24–1 until the target device interrupts.

The following steps correspond to those pictured in Figure 24–1.

❾ IOC$PRIMITIVE_WFIKPCH performs the following steps:

a. It saves register R3 in UCB$Q_FR3 and R4 in UCB$Q_FR4.

b. It sets UCB$V_INT in UCB$L_STS to indicate that an interrupt is expected from this device and UCB$V_TIM to indicate the device interrupt is being timed.

c. It adds the timeout value in R1 to the system uptime, EXE$GL_ABSTIM, and stores the result in UCB$L_DUETIM. This value is the system uptime at which this request will expire.

d. IOC$PRIMITIVE_WFIKPCH clears UCB$V_TIMOUT in UCB$L_STS. If the interrupt does not occur within the stipulated time, EXE$TIMEOUT, in module TIMESCHDL, will set the bit. Chapter *Time Support* describes EXE$TIMEOUT.

e. IOC$PRIMITIVE_WFIKPCH releases the device lock and returns to the driver start I/O routine, which immediately returns to its invoker, EXE$INSIOQ.

Effectively, the WFIKPCH macro invocation saves the context of the driver's start I/O routine in the UCB fork block. The start I/O routine will be resumed after the device interrupt.

❿ EXE$INSIOQ returns to EXE$QIODRVPKT.

⓫ EXE$QIODRVPKT returns control to the I/O requestor, as described in Chapter *I/O System Services*.

24.2.4 Servicing the Device Interrupt

When a device has performed a requested function, it interrupts the processor. Interrupt dispatching, the mechanism by which VMS invokes a device's interrupt service routine (ISR), is described in Section 24.3.

Figure 24–2 shows how the ISR resumes the start I/O routine. The numbers in the figure correspond to the numbered steps that follow.

❶ The common I/O interrupt dispatcher, IO_INTERRUPT, in module IOCINTDISP, saves general registers R0, R1, and R16 through R29 on the stack. It then calls the device's ISR, passing it the address of the interrupt dispatch block (IDB) as an argument.

❷ Example 24–3 shows a simple ISR for a simple driver.

Typically, an ISR must perform the following steps:

a. Obtain the UCB address for the device that interrupted.

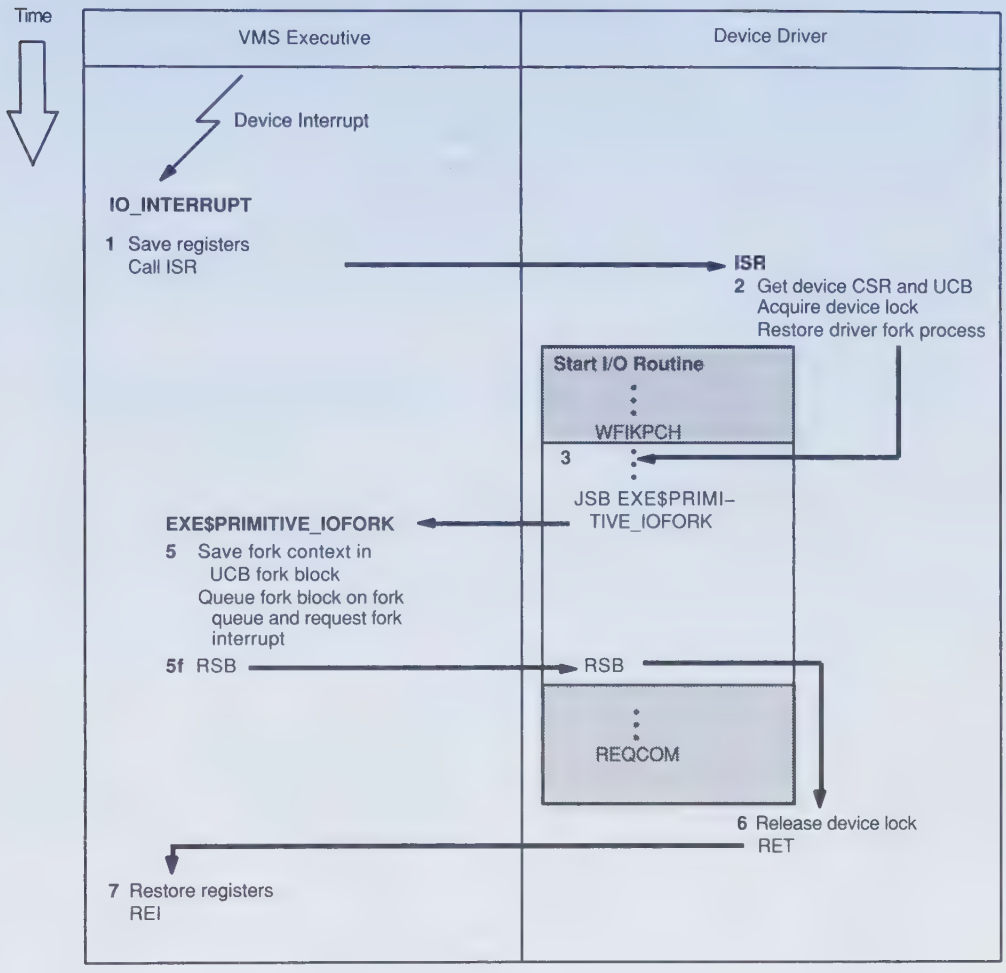
b. Acquire the device lock.

The UCB contains the address of the device lock. The device lock normally synchronizes access to controller and device registers and certain UCB fields. Every code thread that accesses these registers or UCB fields must acquire the device lock first. A driver writer determines which registers and fields the device lock synchronizes based on the nature of the device and the interaction among driver routines like initialization, timeout, start I/O, and interrupt service routines.

c. Test whether the interrupt is expected.

If UCB$V_INT is not set, the interrupt is not expected; the ISR simply releases the device lock and returns to IO_INTERRUPT. Note that for a device such as a terminal, unexpected interrupts are quite normal and would be serviced within its driver.

Figure 24–2 ISR Resumes the Start I/O Routine



Example 24–3 Simple Interrupt Service Routine

```
ISR:    .CALL_ENTRY

        MOVQ    R16, R3                 ;Copy IDB address to R4
        MOVQ    IDB$Q_CSR(R3), R4       ;Get device CSR in R4
        MOVL    IDB$L_OWNER(R3), R5     ;Get owner UCB in R5
        DEVICELOCK -
                LOCKADDR=UCB$L_DLCK(R5) ;Acquire device lock
        BBCC    #UCB$V_INT, -           ;Is interrupt expected?
                UCB$L_STS(R5), 10$      ;If not, go to 10$
        MOVQ    UCB$Q_FR3(R5), R3       ;Otherwise, restore fork R3
        JSB     @UCB$L_FPC(R5)          ;Resume start I/O routine
10$:    DEVICEUNLOCK -                  ;Release device lock
                LOCKADDR=UCB$L_DLCK(R5)
        RET                             ;Return to common interrupt
                                        ; dispatcher
```

d. If the interrupt is expected, the ISR restores fork R3 from the UCB fork block and resumes the start I/O routine at the entry point whose procedure value is in UCB$L_FPC. The processor remains at device IPL. Note that fork R4 is never restored; instead, the second instruction in the ISR stores the device's control/status register (CSR) address in R4.

❸ The procedure value in UCB$L_FPC identifies the entry point L1 in Example 24–2. If the interrupt arrives within the specified time interval, L1 proceeds to step 4; otherwise, it enters DEVTMO.

❹ The start I/O routine performs any device-specific processing that requires execution at device IPL with the device lock held. All other processing is deferred to the fork process.

Since this part of the start I/O routine has been resumed from the ISR at device IPL, it may not directly lower the IPL. The [*censored*] architecture prohibits an interrupt thread of execution from lowering IPL below the level at which it was initiated.

VMS provides the IOFORK macro so that a driver can request the resumption of a thread of execution at a lower IPL. If it is used without any of its optional arguments (that is, in a form compatible with its use in VAX/VMS), the macro expands to the following VAX MACRO code sequence:

```
        MOVAB    L1,FKB$L_FPC(R5)
        JSB      EXE$PRIMITIVE_IOFORK
        RSB
L1:  .JSB_ENTRY INPUT=<R3,R4,R5>,SCRATCH=<R0,R1,R2>
```

❺ EXE$PRIMITIVE_IOFORK, in module FORKCNTRL, assumes that the fork block for the fork process that invoked it is a UCB substructure. It requests the resumption of the fork process at its fork IPL as follows:

a. It clears UCB$V_TIM in UCB$L_STS to indicate that the device no longer has an I/O operation being timed for the delivery of an interrupt.

b. It stores registers R3 and R4 in UCB$Q_FR3 and UCB$Q_FR4.

c. It gets the fork IPL of the fork thread from the array SMP$AL_IPLVEC indexed by the fork lock index in UCB$B_FLCK.

d. EXE$PRIMITIVE_IOFORK locates the head of the fork queue for this fork IPL in the per-CPU database and inserts the fork block into this queue. Chapter *Software Interrupts* describes fork queues.

e. If the fork block is the first to be inserted on the fork queue, EXE$PRIMITIVE_IOFORK requests a software interrupt at fork IPL.

f. It then returns to the start I/O routine, which returns to the ISR.

❻ The ISR releases the device lock and returns to IO_INTERRUPT.

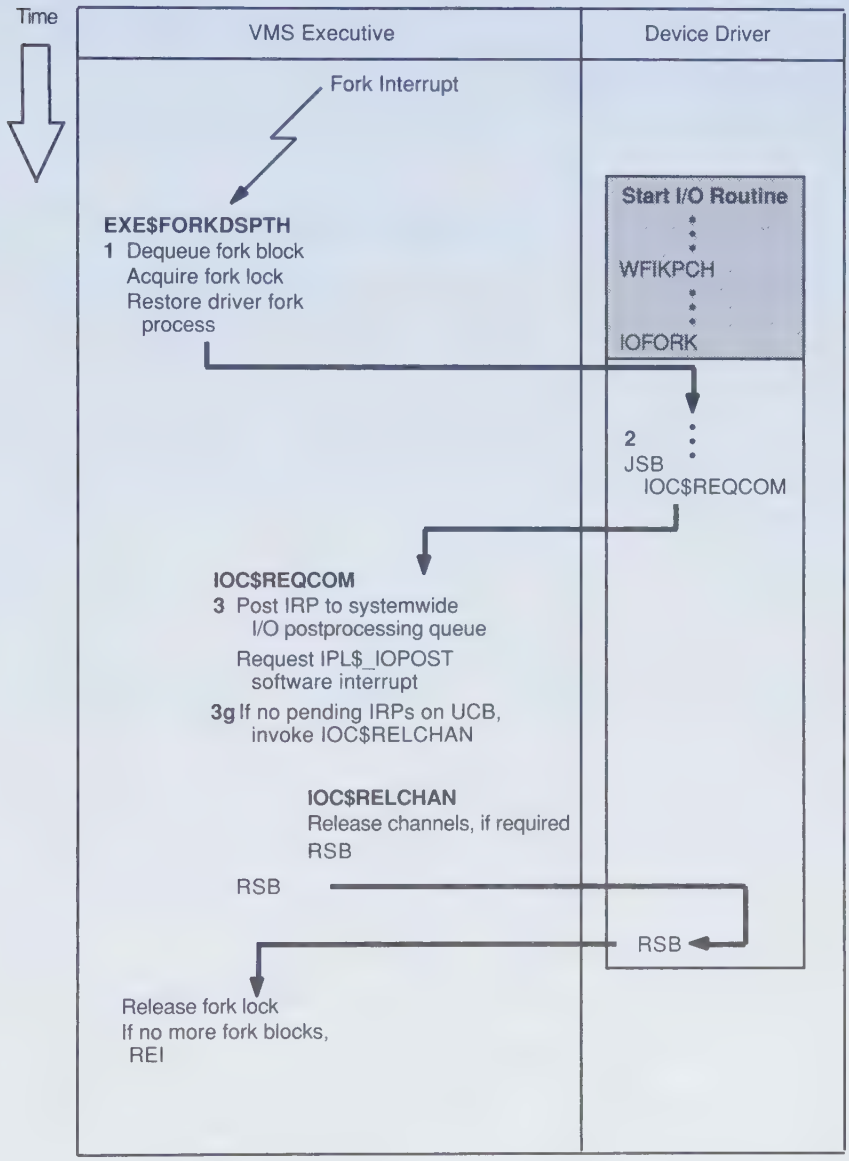
❼ IO_INTERRUPT restores registers R0, R1, and R16 through R29. It then dismisses the interrupt with a `CALL_PAL REI` instruction.

Subsequently reentered at fork IPL, the start I/O routine will resume execution at the VAX MACRO statement following the IOFORK macro.

24.2.5 I/O Request Completion Processing

Figure 24–3 shows the control flow when the fork dispatcher resumes the driver's start I/O routine. The numbers in the figure correspond to the numbered steps in the description that follows.

Figure 24–3 Fork Dispatcher Resumes the Start I/O Routine



When processor IPL falls below the fork IPL of the device, the processor grants the requested software interrupt at that IPL. The software ISR is one of the EXE$FRKIPL*x*DSP routines in module FORKCNTRL, where *x* is 6, 8, 9, 10, or 11, one of the fork IPL values. All these routines converge in EXE$FORKDSPTH, also in module FORKCNTRL.

❶ EXE$FORKDSPTH removes one fork block at a time from the appropriate fork queue and performs the following steps:

a. It acquires the fork lock whose index is in FKB$B_FLCK.

b. It restores R3 and R4 from FKB$Q_FR3 and FKB$Q_FR4 of the fork block. For a UCB, these are the same as offsets UCB$Q_FR3 and UCB$Q_FR4.

c. It invokes the routine whose procedure value is stored in FKB$L_FPC, thus resuming the fork process.

d. When the fork process returns, EXE$FORKDSPTH releases the fork lock.

In Example 24–1, EXE$FORKDSPTH resumes the start I/O routine at the VAX MACRO statement following the invocation of the IOFORK macro.

❷ The start I/O routine performs device-dependent I/O postprocessing, which includes checking for a device error condition. It then constructs in the low-order longwords of registers R0 and R1 the final status of the I/O operation; this is the status to be returned in the I/O requestor's I/O status block.

Upon completion of device-dependent I/O postprocessing, the start I/O routine invokes the REQCOM macro to initiate device-independent I/O postprocessing performed by VMS. The REQCOM macro expands to the following VAX MACRO statements:

```
JSB IOC$REQCOM
RSB
```

❸ IOC$REQCOM, in module IOSUBNPAG, queues the IRP to the systemwide postprocessing queue and requests the IPL$_IOPOST software interrupt. It performs the following steps (details of processing related to mount verification have been omitted):

a. If there is an associated error log buffer (if UCB$V_ERLOGIP in UCB$L_STS is set), IOC$REQCOM transfers the necessary information to the error log buffer and invokes ERL$RELEASEMB, in module ERRORLOG, to complete the error log activity for this I/O operation.

b. It increments the I/O operation count in the UCB.

c. It stores the final I/O status in IRP$L_IOST1 and IRP$L_IOST2.

d. IOC$REQCOM saves the current IPL.

e. IOC$REQCOM inserts the IRP in the interlocked systemwide I/O post-processing queue, IOC$GQ_POSTIQ. It then requests an IPL$_IOPOST interrupt.

f. IOC$REQCOM restores the saved IPL.

g. IOC$REQCOM attempts to remove a pending IRP from the head of the device's queue of pending I/O requests at UCB$L_IOQFL.

If there is one, IOC$REQCOM invokes IOC$INITIATE to initiate the I/O. IOC$INITIATE invokes the driver's start I/O routine.

When that instance of the start I/O routine returns, perhaps after stalling through WFIKPCH to wait for device activity, IOC$INITIATE returns to IOC$REQCOM.

If there are no pending I/O requests for the device, IOC$REQCOM clears UCB$V_BSY in UCB$L_STS and invokes IOC$RELCHAN to release any device controllers to which the start I/O routine had acquired exclusive access.

h. IOC$REQCOM returns to its invoker, the start I/O routine.

❹ The start I/O routine returns to its invoker, which, in this example, is EXE$FORKDSPTH.

24.3 I/O Interrupt Dispatching

The [*censored*] architecture reserves system control block (SCB) vector offsets 800_{16} through $1FF0_{16}$ for I/O interrupts. How these vectors are allocated to devices and actually used is implementation-dependent.

In response to an I/O device interrupt, the processor enters the initiate-exception-or-interrupt (IEI) privileged architecture library (PALcode) routine. The IEI PALcode routine determines the device interrupt vector in a system-specific manner and uses it as the SCB vector offset to locate the interrupt's corresponding SCB entry. Chapter *Interrupts and Exceptions* describes the IEI PALcode routine.

An SCB entry consists of two quadwords. The first quadword of each SCB entry for an I/O interrupt contains the same value: the code address of the common I/O interrupt dispatcher, IO_INTERRUPT. The second quadword contains the address of the interrupt transfer vector (VEC) for the interrupting device. The VEC is usually a substructure of a controller request block (CRB) or an adapter control block (ADP), as described in Section 24.3.2.

PALcode loads R3 with the contents of the second quadword, the VEC address, before entering IO_INTERRUPT. IO_INTERRUPT locates the interrupt's service routine from information in the VEC.

24.3.1 Common I/O Interrupt Dispatcher

Example 24–4 shows a code fragment from IO_INTERRUPT, in module IOCINTDISP, which is written in [*censored*] assembly language.

Running in kernel mode on the interrupted process's kernel stack, IO_INTERRUPT performs the following steps:

1. It saves registers R0, R1, and R16 through R29 on the current stack.
2. IO_INTERRUPT records the contents of VEC$L_IDB(R3) in R16 as an argument for the ISR to be called. VEC$L_IDB(R3) points to an IDB or an ADP, depending on whether the interrupt is directly or indirectly vectored, as described in Section 24.3.2.
3. It uses VEC$PS_ISR_CODE and VEC$PS_ISR_PD to call the ISR.

Example 24–4 Common I/O Interrupt Dispatcher

```
IO_INTERRUPT:
        .
        .
        MB                              ;Issue memory barrier
        LDA     SP,-SAVE_LENGTH(SP)     ;Allocate stack space
        STQ     R0,SAVR0(SP)            ;Save registers R0, R1,
        .                               ; R16-R29
        .                               ;
        STQ     R29,SAVR29(SP)          ;
        LDL     R16,VEC$L_IDB(R3)       ;Set up parameter to driver
        LDA     R25,1(R31)              ;One argument
        LDL     R26,VEC$PS_ISR_CODE(R3) ;Get code address of ISR
        LDL     R27,VEC$PS_ISR_PD(R3)   ;Get ISR's procedure
                                        ; value
        JSR     R26,(R26)               ;Call the ISR
        LDQ     R0,SAVR0(SP)            ;Restore registers R0, R1,
        .                               ; R16-R29
        .                               ;
        LDQ     R29,SAVR29(SP)          ;
        LDA     SP,SAVE_LENGTH(SP)      ;Restore stack pointer
        REI                             ;Now dismiss the interrupt
```

4. When the ISR returns control, IO_INTERRUPT restores the saved registers and executes a CALL_PAL REI instruction to dismiss the interrupt.

24.3.2 Directly and Indirectly Vectored Interrupts

Device interrupt requests are conveyed to a [*censored*] processor from two sources: local I/O widgets and local I/O adapters. Interrupt requests generated by devices attached to remote I/O widgets are conveyed to the processor through one or more intervening I/O adapters.

If VMS can directly dispatch a device interrupt to the interrupting device's ISR, the interrupt is said to be directly vectored; otherwise, it is indirectly vectored.

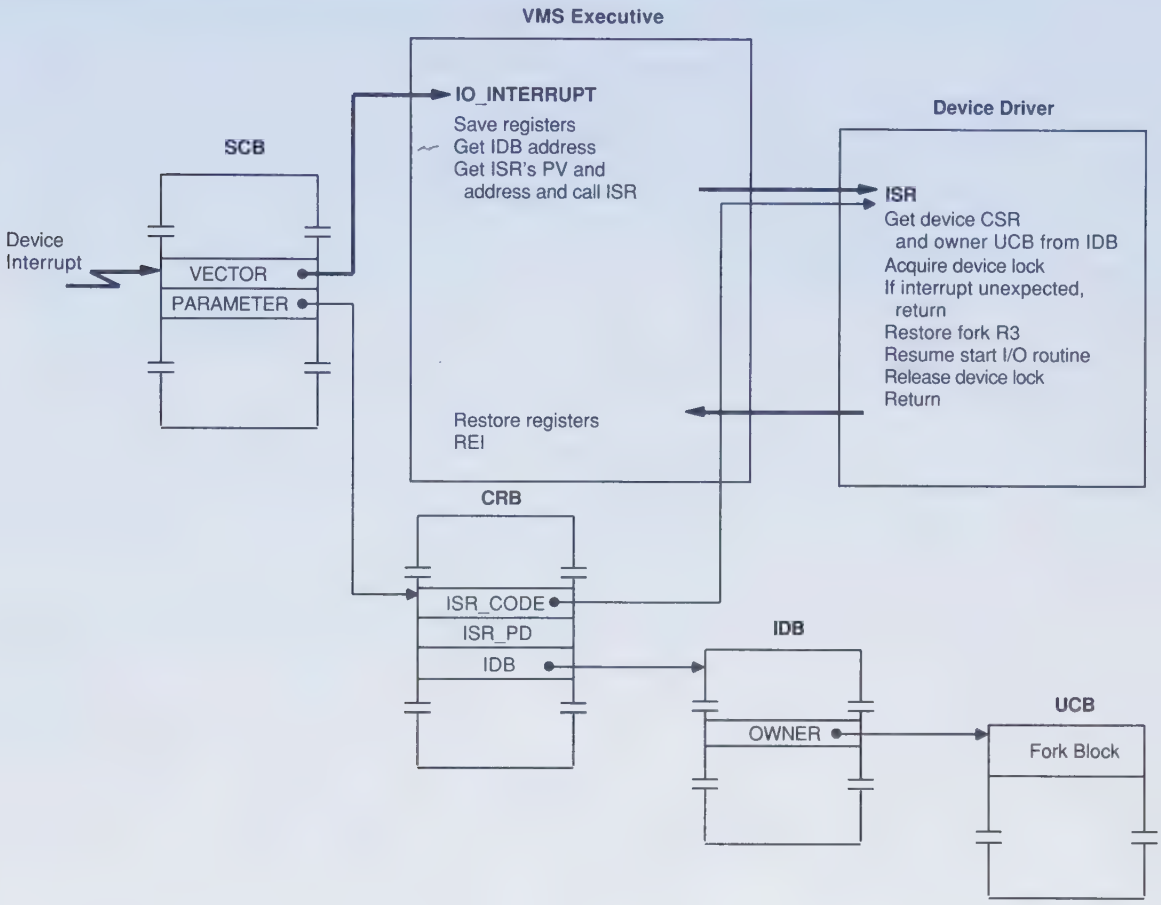
Whether VMS can directly vector a device interrupt depends primarily on system and PALcode implementation. If PALcode cannot directly determine an interrupting device's vector, the device's interrupts cannot be directly vectored. Such a device's interrupts are vectored through an intermediate interrupt service routine, typically that of the local I/O adapter or multichannel I/O widget that conveys the device's interrupt requests to the processor.

When PALcode can directly determine an interrupting device's vector, VMS can implement either method of vectoring, depending on other system-specific factors.

Whether a device interrupt is directly vectored is transparent to IO_INTERRUPT. IO_INTERRUPT uses the area pointed to by R3 as a VEC substructure.

For a device with directly vectored interrupts, the VEC is a CRB substructure, and VEC$L_ISR_CODE and VEC$L_ISR_PD describe the interrupting device's ISR. The control flow from IO_INTERRUPT to the device's ISR is shown in Figure 24–4.

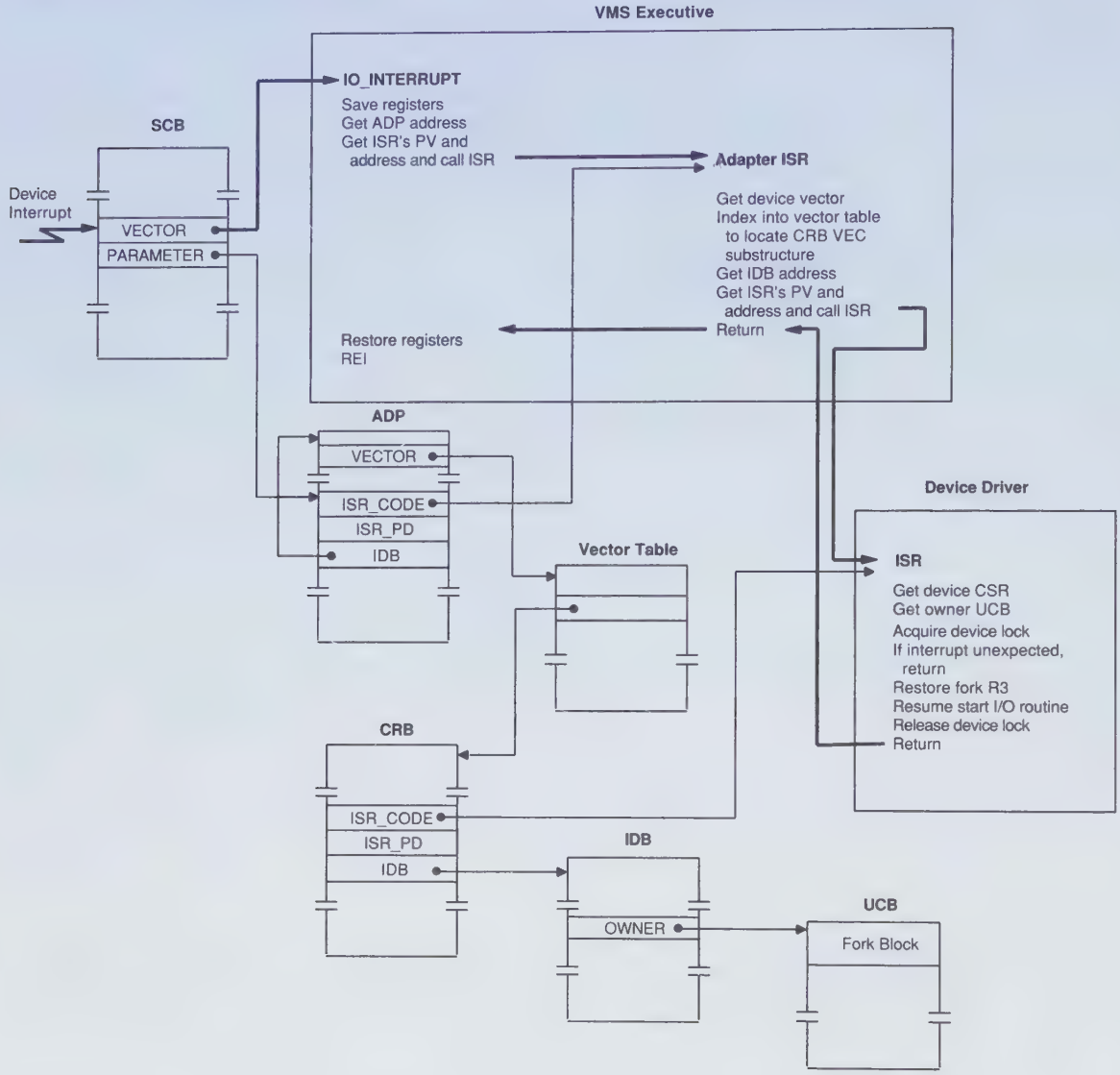
Figure 24–4 Control Flow for a Directly Vectored Interrupt



For a device with indirectly vectored interrupts, the VEC is a substructure of the ADP for the local I/O adapter or multichannel I/O widget that conveys the device's interrupt request to the processor; VEC$L_ISR_PD and VEC$L_ISR_CODE describe the ISR for the local I/O adapter or multichannel I/O widget.

The control flow from IO_INTERRUPT to the device's ISR for an indirectly vectored interrupt is shown in Figure 24–5. IO_INTERRUPT calls the adapter's ISR with the ADP address as an argument (see Section 24.3.1). The adapter ISR determines the interrupting device's vector in a device- and interconnect-specific manner. It then uses the interrupt vector as an index into the adapter's interrupt vector table. The interrupt vector table, pointed to by ADP$L_VECTOR, is an array of pointers to VEC substructures. Indexed by device interrupt vector, an entry in the array points to the CRB VEC substructure for a controller whose interrupts are indirectly vectored through the adapter's ISR. The adapter's ISR calls the interrupting device's ISR using information in the appropriate CRB VEC substructure.

Figure 24–5 Control Flow for an Indirectly Vectored Interrupt



24.4 Driver Kernel Processes

As explained in Section 24.2, for every I/O that VMS initiates, it creates a fork process thread of execution in which driver code executes. A fork process's context is maintained in the fork block, which is usually part of a larger data structure such as the UCB or the IRP. A driver fork process that requires more context can execute as a kernel process. A kernel process has a private stack on which it can maintain context across stalls and restarts.

Chapter *Software Interrupts* describes fork processes and kernel processes in detail.

A VAX MACRO driver being ported from VAX/VMS to *censored* VMS can be converted to use the kernel process mechanism through macros provided by *censored* VMS. In general, these macros are named KP_STALL_*xxx* or KP_*xxx*, where *xxx* is the name of the equivalent VAX/VMS macro, if one exists.

Example 24–5 shows the simple driver's start I/O routine of Example 24–1, modified to use the kernel process mechanism. To use the kernel process mechanism, a VAX MACRO device driver must follow certain conventions. The numbers in the following list correspond to those in Example 24–5.

❶ The DDTAB macro invocation must identify EXE$KP_STARTIO as the START argument and the start I/O routine within the driver as the KP_STARTIO argument.

❷ The start I/O routine within the driver must be a standard-conforming procedure.

❸ The start I/O procedure must retrieve the addresses of the IRP and UCB from the kernel process block (KPB) associated with the kernel process.

❹ The start I/O procedure must use the KP_STALL_*xxx* or KP_*xxx* macros instead of equivalent VAX/VMS macros.

Example 24–5 Simple Start I/O Routine That Uses the Kernel Process Mechanism

```
          .
          .
        DDTAB -
          .
          .
              START=EXE$KP_STARTIO,- ❶
              KP_STARTIO=STARTIO,-        ;Miscellaneous other
                                          ; required changes ignored
          .
          .
STARTIO:  .CALL_ENTRY <R2,R3,R4,R5>  ❷
          MOVL    4(AP),R0                ;Get KPB address
          MOVL    KPB$PS_UCB(R0),R5   ❸   ;Get UCB address
          MOVL    KPB$PS_IRP(R0),R3       ;Get IRP address
          .
          .
          KP_STALL_WFIKPCH DEVTMO,#6  ❹   ;Wait for interrupt
          .                               ; or timeout
          .
          KP_STALL_IOFORK                 ;Wait until IPL drops
          .                               ; to fork IPL
          .
          KP_REQCOM                       ;Complete request
```

What follows is a brief description of the control flow of an I/O operation through the start I/O procedure of Example 24–5. Although the details of the interaction between the start I/O procedure and VMS are quite different from the interaction described in Section 24.2, the overall structure of a driver that uses the kernel process mechanism is much the same as that of a driver using the fork process mechanism.

The vertical bars in the rightmost column of Figures 24–6, 24–7, and 24–8 represent stack execution: the right-hand bar, execution on the kernel process's stack, and the left-hand bars, execution on another kernel stack.

24.4.1 Driver Kernel Process Startup

Control flow from EXE$QIO to IOC$INITIATE is identical for drivers using the kernel process mechanism and drivers using the fork process mechanism. IOC$INITIATE locates the driver's start I/O routine and invokes it; in this example, it invokes EXE$KP_STARTIO, the routine identified by the DDTAB macro START argument.

EXE$KP_STARTIO performs the following steps to create a kernel process thread of execution running procedure STARTIO. The numbers in Figure 24–6 correspond to those in the following list:

❶ EXE$KP_STARTIO checks whether the IRP is one that is being used by the modified page writer; if so, it goes to step 4, skipping the KPB allocation because the modified page writer uses preallocated KPBs.

❷ It computes the kernel process's required stack size as the larger of KPB$K_MIN_IO_STACK and DDT$IS_STACK_BCNT and calls EXE$KP_ALLOCATE_KPB to allocate a KPB and that much stack.

❸ If EXE$KP_ALLOCATE_KPB returns a success status, EXE$KP_STARTIO proceeds to the next step.

If EXE$KP_ALLOCATE_KPB returns the failure status SS$_INSFRPGS (insufficient free pages) or SS$_INSFMEM (insufficient memory), EXE$KP_STARTIO calls EXE$KP_ALLOCATE_KPB once a second until the operation is successful or the I/O request is canceled. If the I/O request is canceled, EXE$KP_STARTIO completes the request by setting the completion status to SS$_ABORT and invoking the REQCOM macro. Upon successful return from EXE$KP_ALLOCATE_KPB, EXE$KP_STARTIO proceeds to the next step.

If EXE$KP_ALLOCATE_KPB returns any other error status, EXE$KP_STARTIO generates the fatal bugcheck INCONSTATE.

❹ EXE$KP_STARTIO initializes KPB$PS_IRP and KPB$PS_UCB with the IRP and UCB addresses.

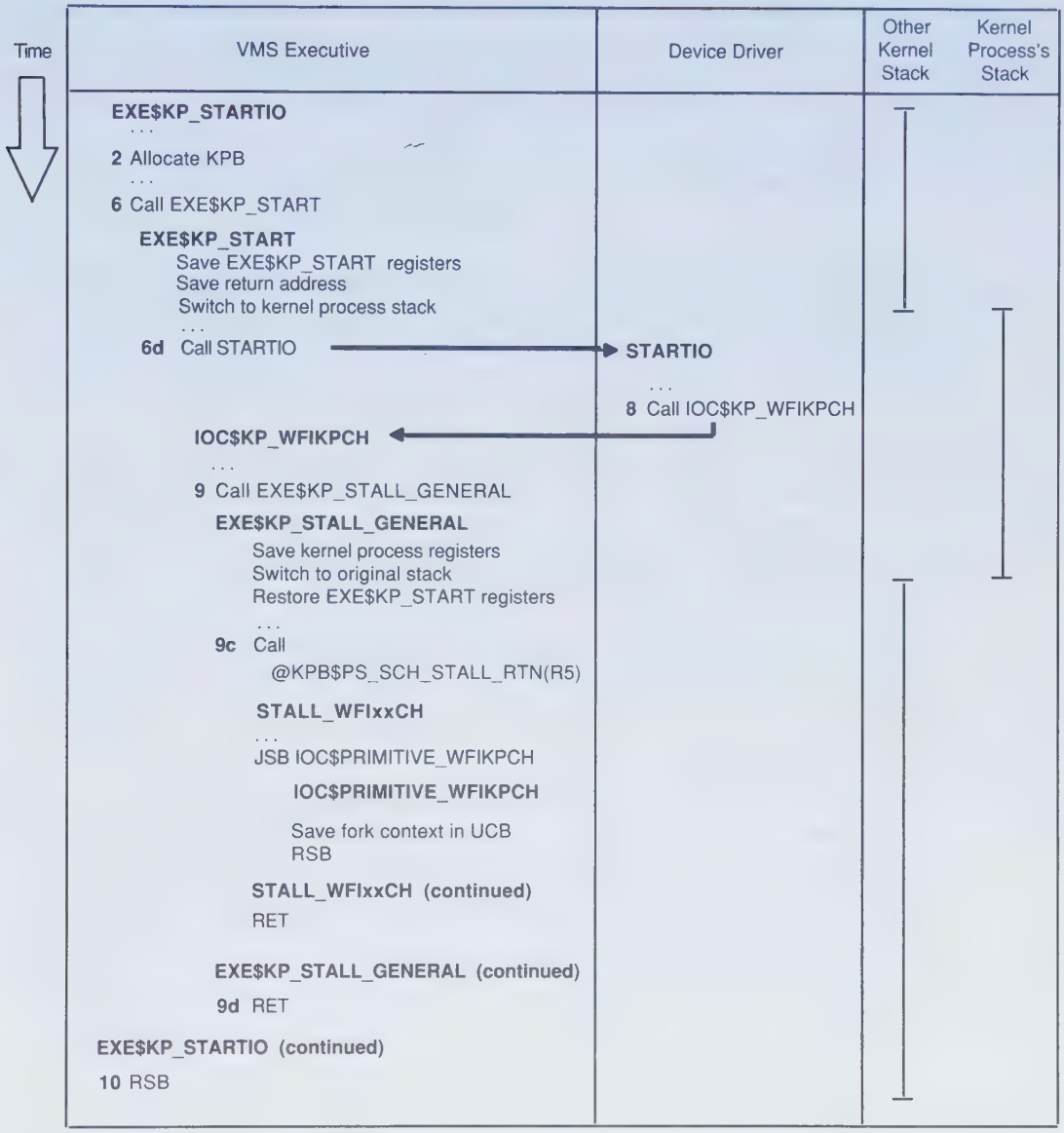
❺ It copies STARTIO's register save mask from DDT$IS_REG_MASK into R0. It then performs some logical operations on R0 to create a register save mask that includes all registers that kernel process support routines need saved but none that need not be saved (R0, R1, R16–R25, R27, R28, R30, R31).

❻ EXE$KP_STARTIO calls EXE$KP_START, in module KERNEL_PROCESS_MAGIC. EXE$KP_START starts a driver kernel process thread of execution by taking the steps summarized in the following list (see Chapter *Software Interrupts* for further details):

a. EXE$KP_START saves the registers specified in R0's register save mask on the current stack.

b. It saves the current stack pointer in KPB$PS_SAVED_SP.

Figure 24–6 Driver Kernel Process Startup



c. It switches to the kernel process's stack by loading the stack pointer (SP) from KPB$PS_STACK_BASE.

d. It calls STARTIO, the procedure whose procedure value is in DDT$PS_KP_STARTIO, with the KPB address as the single argument.

❼ STARTIO loads R3 and R5 from the IRP and UCB addresses in the KPB. It then acquires the device lock and initiates device activity.

❽ After initiating device activity, STARTIO invokes the macro KP_STALL_WFIKPCH, which, for the given example, expands as shown in Example 24–6.

Example 24–6 Expansion of the KP_STALL_WFIKPCH Macro

```
;Expansion of KP_STALL_WFIKPCH DEVTMO,#6

                                ;Top of stack has IPL to
                                ; restore after setting up wait
PUSHL     #6                    ;Timeout value
PUSHL     KPB                   ;KPB address
CALLS     #3,IOC$KP_WFIKPCH     ;
BLBC      R0,DEVTMO             ;If operation timed out,
                                ; enter timeout routine
```

❾ IOC$KP_WFIKPCH, in module KERNEL_PROCESS, validates its arguments and copies them to the KPB. It records the procedure value of STALL_WFI*xx*CH, also in module KERNEL_PROCESS, in KPB$PS_SCH_STALL_RTN and calls EXE$KP_STALL_GENERAL, in module KERNEL_PROCESS_MAGIC, to stall the kernel process.

The actions of EXE$KP_STALL_GENERAL are detailed in Chapter *Software Interrupts*. Steps that are relevant to this control flow are summarized in the following list:

a. EXE$KP_STALL_GENERAL saves the kernel process's context on the kernel process's stack.

b. It restores the stack and register contexts that were current when the kernel process was entered.

c. It calls STALL_WFI*xx*CH (the routine whose procedure value is in KPB$PS_SCH_STALL_RTN).

   STALL_WFI*xx*CH invokes the WFIKPCH macro, specifying the macro statement RET as the return mechanism. The WFIKPCH macro invocation generates an entry point in STALL_WFI*xx*CH and stores its procedure value in UCB$L_FPC. It then invokes IOC$PRIMITIVE_WFIKPCH, which records the fork context of the driver kernel process, releases the device lock, and returns to STALL_WFI*xx*CH. STALL_WFI*xx*CH returns to EXE$KP_STALL_GENERAL.

d. EXE$KP_STALL_GENERAL loads the success status SS$_NORMAL in R0 and returns to the routine whose return address was saved on the kernel stack, which, for this example, is EXE$KP_STARTIO.

❿ When control returns from EXE$KP_STALL_GENERAL, EXE$KP_STARTIO tests the status in R0. If R0 contains a success status, EXE$KP_STARTIO returns to its invoker, which, in this example, is IOC$INITIATE. If R0 contains an error, EXE$KP_START was unable to start the kernel process for some reason and EXE$KP_STARTIO generates the fatal bugcheck INCONSTATE.

The control flow from IOC$INITIATE back to the $QIO requestor is the same as that for a driver using fork processes.

24.4.2 Device Interrupt Resumes Driver Kernel Process

Figure 24–7 shows the control flow when the device activity completion interrupt resumes the driver kernel process. The numbers in the following list correspond to those in Figure 24–7. Most of the details are left out of the steps here because they are given elsewhere in this chapter or in Chapter *Software Interrupts*.

❶ When the device interrupts, HEI PALcode invokes IO_INTERRUPT, as discussed in Section 24.3.2.

❷ IO_INTERRUPT calls the device's ISR.

❸ At step 9c in Section 24.4.1, STALL_WFI*xx*CH invoked the WFIKPCH macro. The WFIKPCH macro invocation generated an entry point in STALL_WFI*xx*CH, and stored its procedure value in UCB$L_FPC. The device's ISR obtains the device lock and resumes STALL_WFI*xx*CH at this entry point.

❹ STALL_WFI*xx*CH calls EXE$KP_RESTART.

❺ EXE$KP_RESTART saves the register context of its caller, switches to the kernel process's stack, and restores the kernel process's registers. The most recent call frame on the kernel process's stack was left there when the driver kernel process earlier called IOC$KP_WFIKPCH. EXE$KP_RESTART returns to the STARTIO procedure from its call to IOC$KP_WFIKPCH.

❻ The STARTIO procedure performs device-specific status checks of the I/O operation that just completed. It performs only the steps that must be performed at device IPL, before invoking the KP_STALL_IOFORK macro to resume the kernel process at the lower fork IPL. The KP_STALL_IOFORK macro expands as follows:

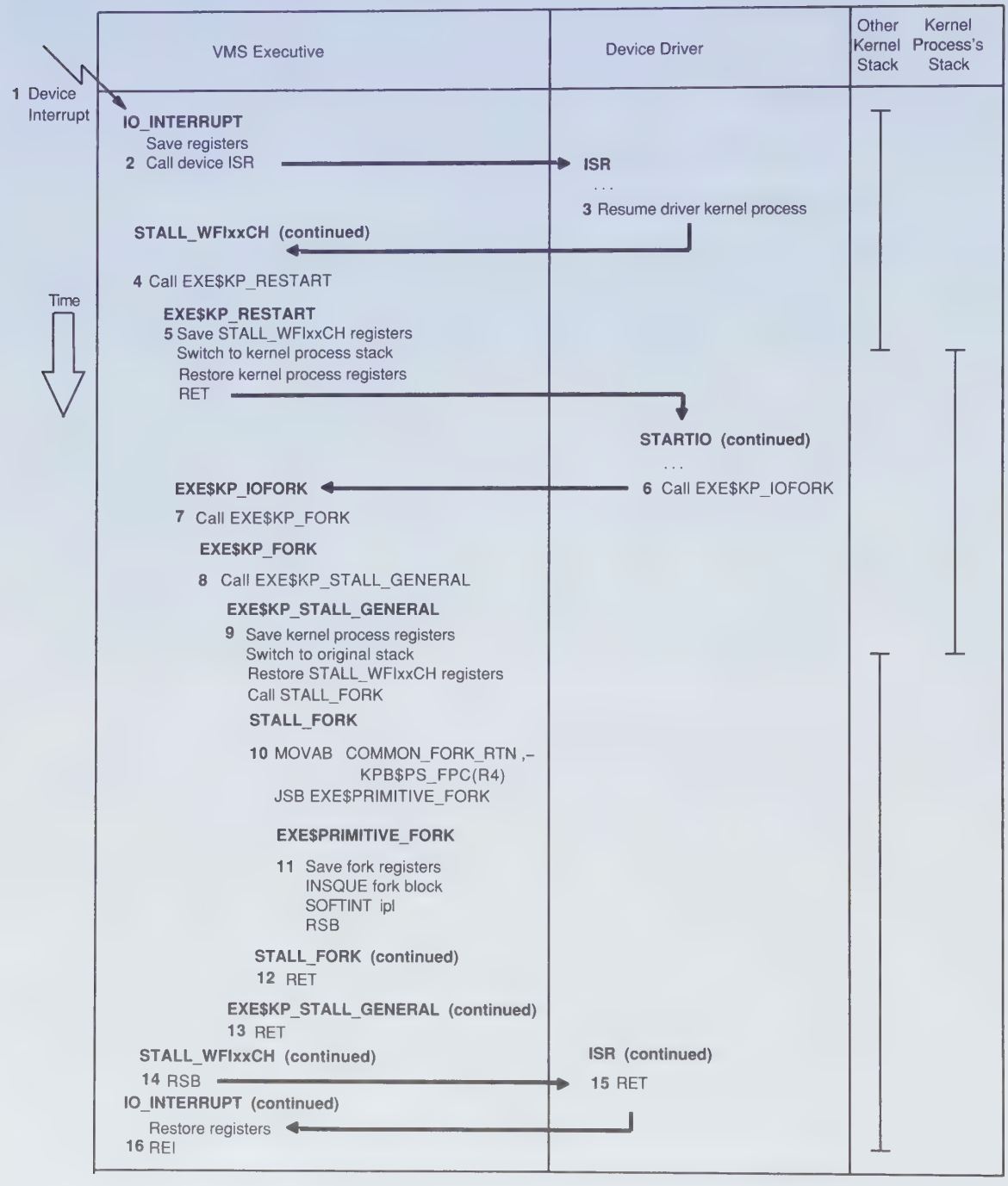
```
PUSHL IRP$PS_KPB(R3)
CALLS #1,EXE$KP_IOFORK
```

❼ EXE$KP_IOFORK clears UCB$V_TIM in UCB$L_STS to indicate that the device is no longer being timed for I/O and calls EXE$KP_FORK.

❽ EXE$KP_FORK saves the kernel process's fork context in the UCB fork block. It places the procedure value of STALL_FORK, also in module KERNEL_PROCESS, into KPB$PS_SCH_STALL_RTN and calls EXE$KP_STALL_GENERAL.

❾ EXE$KP_STALL_GENERAL saves the kernel process's register context in the KPB, switches to the original kernel stack and restores the registers that were saved in step 5, when the kernel process was resumed. It then calls STALL_FORK, the procedure whose procedure value is in KPB$PS_SCH_STALL_RTN.

Figure 24–7 Device Interrupt Resumes Driver Kernel Process



⑩ STALL_FORK stores the procedure value of COMMON_FORK_RTN, also in module KERNEL_PROCESS, in KPB$PS_FPC, and invokes EXE$PRIMITIVE_FORK.

⑪ EXE$PRIMITIVE_FORK saves the fork registers R3 and R4 in the UCB fork block, inserts the UCB fork block into the appropriate fork queue, requests a fork IPL interrupt if appropriate, and returns to STALL_FORK.

⑫ STALL_FORK returns to its caller, EXE$KP_STALL_GENERAL.

⑬ At this point, the most recent call frame on the original kernel stack is the one left there by STALL_WFIxxCH when it called EXE$KP_RESTART. EXE$KP_STALL_GENERAL returns to STALL_WFIxxCH.

⑭ STALL_WFIxxCH returns to the ISR.

⑮ The ISR releases the device lock and returns to IO_INTERRUPT.

⑯ IO_INTERRUPT restores the registers it saved and dismisses the interrupt with a CALL_PAL REI instruction.

24.4.3 Fork Interrupt Resumes Driver Kernel Process

Figure 24–8 shows the control flow when the fork IPL software interrupt resumes the driver kernel process. The numbers in the following list correspond to those in Figure 24–8. Most of the details are left out of the steps here because they are given elsewhere in this chapter or in Chapter *Software Interrupts*.

❶ When processor IPL drops below the fork IPL, the fork IPL software interrupt is granted. The fork dispatcher ISR, EXE$FRKIPLxDSP, where *x* is 6, 8, 9, 10, or 11, one of the fork IPLs, is entered. This example assumes a fork IPL of 8.

❷ EXE$FRKIPL8DSP obtains the offset to the IPL 8 fork queue listhead (see Chapter *Software Interrupts*) and enters EXE$FRKDSPTH.

❸ EXE$FRKDSPTH is a common entry point used by all fork IPL ISRs. It resumes pending fork processes by performing the following steps:

a. It removes a fork block from the fork queue. If no fork block was removed, it dismisses the fork IPL interrupt using the CALL_PAL REI instruction.

b. It acquires the fork lock whose index is in FKB$B_FLCK.

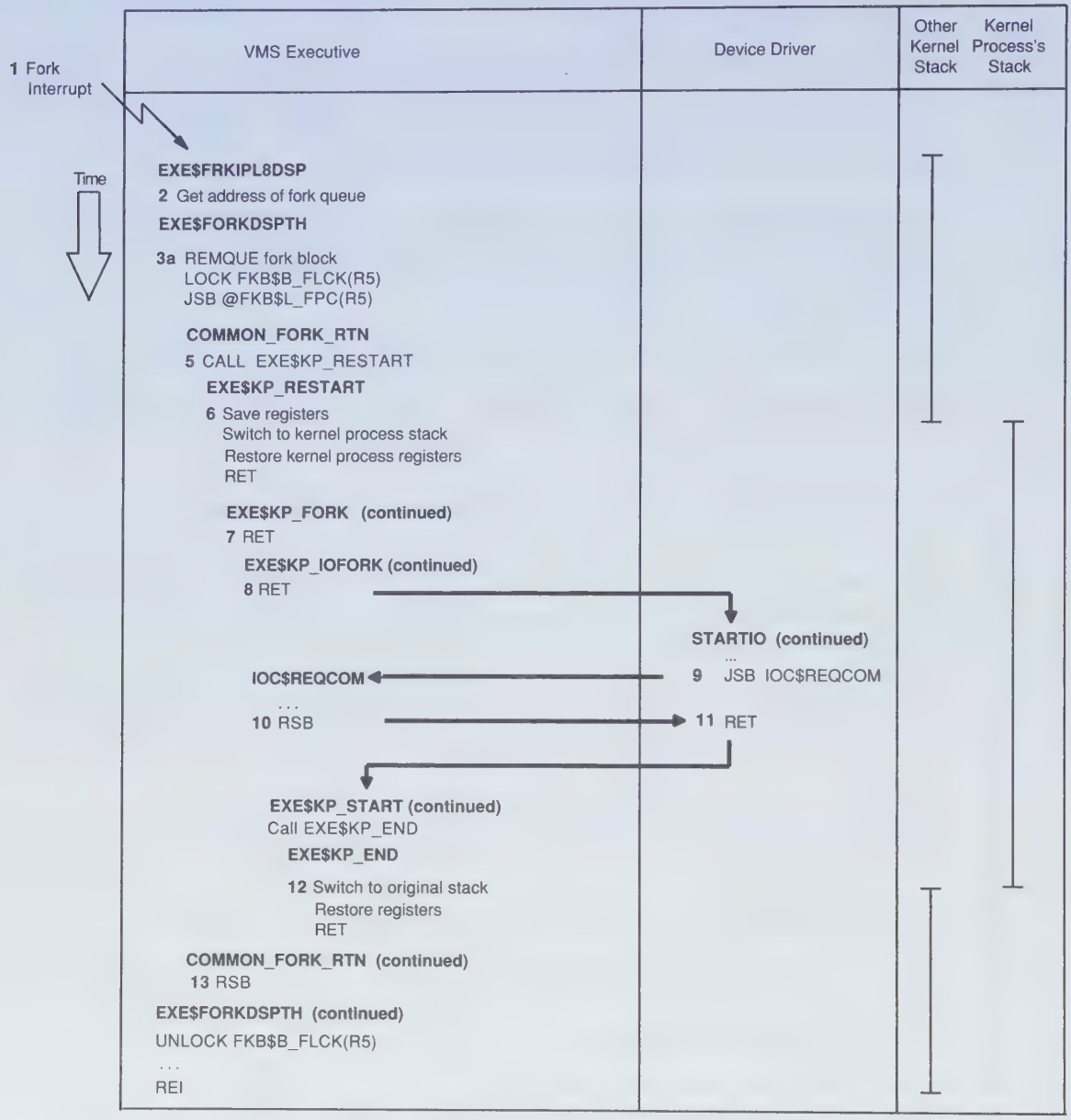
c. It resumes the fork process.

❹ The fork process invokes COMMON_FORK_RTN.

❺ COMMON_FORK_RTN calls EXE$KP_RESTART.

❻ EXE$KP_RESTART saves the fork process's register context on the current stack. R4 contains the KPB address of the kernel process that must be resumed. EXE$KP_RESTART switches to the kernel process's stack, restores the kernel process's registers, and resumes the kernel process by executing the VAX MACRO statement RET.

Figure 24–8 Fork Interrupt Resumes Driver Kernel Process



The most recent call frame on the kernel process's stack is the one left by EXE$KP_FORK when it earlier called EXE$KP_STALL_GENERAL. Thus the RET statement resumes EXE$KP_FORK.

❼ EXE$KP_FORK returns to its caller, EXE$KP_IOFORK.

❽ EXE$KP_IOFORK returns to its caller, the STARTIO procedure.

❾ The STARTIO procedure completes device-specific I/O postprocessing and invokes the KP_REQCOM macro. The KP_REQCOM macro expands to the following VAX MACRO statements:

```
JSB IOC$REQCOM
RET
```

❿ After IOC$REQCOM performs the actions detailed in Section 24.2.5, it returns to the STARTIO procedure.

⓫ At this point, the most recent call frame on the kernel process's stack is the one left there by EXE$KP_START when it earlier started up the driver kernel process and called the STARTIO procedure (see step 6d, Section 24.4.1). STARTIO returns to EXE$KP_START. EXE$KP_START calls EXE$KP_END to end the kernel process.

⓬ At this point, the most recent call frame on the original kernel stack is the one left there by COMMON_FORK_RTN when it earlier called EXE$KP_RESTART. EXE$KP_END switches to the original kernel stack, restores registers that were saved by EXE$KP_RESTART, and returns to COMMON_FORK_RTN.

⓭ COMMON_FORK_RTN returns to EXE$FORKDSPTH, which releases the fork lock and proceeds to step 3a.

24.5 Revision Information

Source Module	Build Date
[SYS]IOSUBNPAG.MAR	9-Aug-1991
[SYS]IOCINTDISP.M64	9-Aug-1991
[SYS]FORKCNTRL.MAR	9-Aug-1991
[SYS]KERNEL_PROCESS.MAR	9-Aug-1991

Part VIII/Life of the System

Chapter 32

The Modular Executive

Non sunt multiplicanda entia praeter necessitatem.
[Entities should not be multiplied beyond necessity.]

William of Occam

The VMS executive consists of two base images and a number of separately loadable executive images. Some of these images are loaded on all systems, while others support features unique to particular system configurations.

The base images, SYS$PUBLIC_VECTORS.EXE and SYS.EXE, connect requests for system services and other system functions with the routines that provide them. These routines are located in separately loadable executive images.

This chapter describes the organization of executive images and the connections among them. It also discusses how executive images are loaded and initialized.

32.1 Overview

The organization of the [*censored*] VMS executive is based upon that of the VAX/VMS executive. In early versions of VAX/VMS, much of the executive was in SYS.EXE, the system image, and RMS.EXE, the image containing Record Management Services. Features not common to all system configurations were supported in separate images, such as device drivers and the SYSLOA*xxx*.EXE images. With this relatively monolithic design, changing the executive often meant applying complex patches to SYS.EXE or rebuilding it. Rebuilding had the undesirable side effect of requiring a subsequent relinking of all images, both VMS- and user-supplied, that linked against SYS.STB to resolve SYS.EXE references.

The VAX/VMS Version 5 executive was further partitioned to simplify subsequent changes to the executive and to reduce the number of system-dependent images that would require relinking when the executive changed. This partitioned executive came to be known as the modular executive.

The concept underlying the VAX/VMS modular executive is similar to that of a VAX/VMS shareable image, which contains transfer vectors and routines. The VAX/VMS system image has been split into a base image named SYS.EXE and a number of other images called executive images. Unlike transfer vectors for a standard shareable image, which are in the same image as the routines, all the transfer vectors of executive images are collected in SYS.EXE. SYS.EXE

also contains systemwide data cells that are referenced from multiple executive images and pointers to data cells in executive images that are referenced from other executive images. The executive images contain the routines to which the transfer vectors dispatch as well as data cells primarily referenced from within the image.

Like the VAX/VMS executive, the [*censored*] VMS executive is modular. The concept underlying its organization is also that of a shareable image, although [*censored*] shareable images differ somewhat from VAX shareable images. A [*censored*] shareable image contains a global symbol table and a symbol vector. The global symbol table lists the name of each universal symbol in the image along with the offset of its corresponding entry in the symbol vector. Although part of the image file, the global symbol table is not loaded into memory. The linker uses it to resolve symbolic references when another image links to the shareable image.

The symbol vector contains an entry for each universal symbol. For a universal symbol that is the name of a data cell, the symbol vector entry contains the address of the data cell. For a universal symbol that is the name of a procedure, the symbol vector entry contains a pair of addresses: the address of the procedure's code entry point and the address of its procedure descriptor. As long as the universal symbol's offset in the symbol vector remains constant across shareable image rebuilds, images referencing the symbol do not need to be rebuilt. When the shareable image is activated, the symbol vector is loaded into memory and the addresses in its entries relocated to point to the actual locations of the loaded code and data. When another image is activated that references universal symbols in the shareable image, its references are fixed up with the contents of the symbol vector entries for those symbols.

Section 32.4.2 describes extensions to the implementation of shareable images required for executive images.

The [*censored*] VMS executive is implemented as two base images, SYS$PUBLIC_VECTORS.EXE and SYS.EXE, and a number of separately loadable executive images. These images are loaded during system initialization: SYS$PUBLIC_VECTORS.EXE from directory SYS$LIBRARY and the others from directory SYS$LOADABLE_IMAGES. Most are permanently mapped, although a particular image can be identified at load time as *removable* and later unloaded.

The base images contain the symbol vectors against which images invoking executive routines, requesting system services, and referencing system data cells link.

An executive routine executes in the mode from which it is invoked, typically executive or kernel mode. Executive images and inner access mode procedures that execute in privileged processes invoke executive routines to perform various functions, for example, to allocate nonpaged pool.

The executive images that contain the executive routines and system service procedures have no symbol vectors. Instead, their universal symbols are represented in the symbol vector in the base image SYS.EXE. SYS.EXE also contains data, pointers to data in executive images, and some primitive routines used by most executive images. Section 32.2 describes SYS.EXE in more detail.

SYS$PUBLIC_VECTORS.EXE is the pathway to system services. A system service is an executive procedure that performs a service for a process. Generally, the service is requested from an outer access mode, although the procedure that performs the service executes in an inner mode. The symbol vector in SYS$PUBLIC_VECTORS.EXE contains entries for the universally available symbols that represent system service names. These symbols typically are the names of transfer routines, minimal procedures in SYS$PUBLIC_VECTORS.EXE that change access mode and dispatch to the actual service procedures in executive images. Section 32.3 presents an overview of SYS$PUBLIC_VECTORS.EXE.

In general, neither base image needs to be rebuilt when corrections or changes are made to executive images. It is possible to replace an executive image with a corrected or enhanced one without any impact on SYS.EXE. However, if universally available entry points or data cells are removed or added, the base image whose symbol vector contains the affected entry points may need to be rebuilt. Note that removals and additions should be made so as not to change the symbol vector offsets of valid symbols.

The modular organization of the executive also makes it possible to have alternative versions of an executive image and to select the one to be loaded at system initialization. For example, there are three versions of the system synchronization image; during system initialization, the version appropriate to the configuration is selected and loaded.

Executive images are loaded much like user images. During system operation, a user image is mapped into address space by the Image Activate ($IMGACT) system service, commonly known as the image activator. The image's references to addresses within and outside itself are relocated and fixed up by the Address Relocation Fixup ($IMGFIX) system service. The term *relocation* means the adjustment of absolute address pointers in an image that refer to locations in that image. An image's absolute addresses cannot be relocated before the image's sections have been mapped. The term *fixup* means the adjustment of absolute address pointers in an image that refer to locations in another image. An image's references to locations in another image cannot be fixed up until the other image's sections have been mapped and the addresses in its symbol vector entries relocated.

After these two system services, whose operations are described in Chapter *Image Activation and Exit*, prepare an image to execute, the image initialization procedure, if any, is called. As the image executes, its pages are read into memory in response to page faults.

Similar steps are required to prepare an executive image to execute. There are, however, several key differences:

- Executive images are mapped into system space.
- Although the pageable pages of an executive image are read into memory in response to page faults, the nonpageable pages are read into memory as part of the loading process.

- Like absolute references in any image, those in an executive image must be relocated and fixed up. Many executive images are loaded during a stage of system initialization when no page faults are allowed. Nonpageable sections' references are relocated and fixed up when the image is loaded; relocation and fixup of references within a pageable section may have to be deferred until paging is possible.
- The contents of SYS.EXE symbol vector entries that represent universal symbols in the executive image must be replaced by addresses within the executive image.
- An executive image's initialization procedure may require that other executive images' initialization procedures have executed or that a particular stage of system initialization has been reached. When an initialization procedure is called, it checks whether the environment is suitable for its execution and, if not, returns a status indicating that it must be called again at a later stage.

These steps are described throughout the rest of this chapter.

32.2 The Base Image SYS.EXE

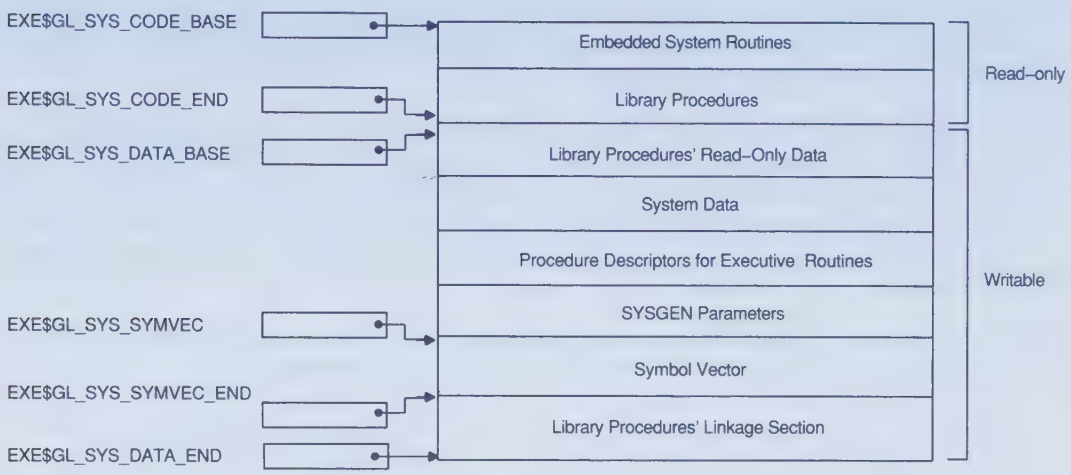
SYS.EXE is loaded by SYSBOOT, the secondary bootstrap program (see Chapter *Bootstrap Processing*). Most important, SYS.EXE contains a symbol vector whose entries represent universally available routines and data in other executive images and in itself. It contains data accessed by many other executive images and a small amount of executable code: several small routines and some library procedures used by most executive images.

SYS.EXE is the pathway to routines and data in other executive images. All images that refer to executive universal symbols link with SYS.EXE. For example, each executive image is linked with SYS.EXE to resolve references to routines and data in other executive images and in SYS.EXE. [*Censored*] VMS and VAX/VMS differ in this regard: a VAX/VMS image links with SYS.STB to resolve SYS.EXE references.

The offsets of entries in SYS.EXE's symbol vector are guaranteed to remain the same in subsequent versions of [*censored*] VMS. It is thus possible that an image linked with SYS.EXE can continue to execute on a later version than the one with which it linked. Whenever algorithmic or data structure changes in an executive subsystem might prevent correct execution of an image that referenced the earlier version, the version category associated with that subsystem is updated. An image linked with an earlier version of that subsystem cannot then be activated on a system running the updated version. Section 32.8 describes subsystem version categories.

Figure 32–1 shows the organization of SYS.EXE, which is defined at link time. Entirely nonpageable, SYS.EXE consists of a single read-only image section and a single writable image section. As shown in the figure, data cells within it (in module SYSTEM_DATA_CELLS) are initialized at load time to point to the loaded base and end of some of its areas.

Figure 32–1 Layout of SYS.EXE



Its image sections are made up of the following areas:

- Several small system routines, referred to here as embedded system routines
- Library procedures and their associated read-only data and linkage section
- Commonly accessed writable data and pointers to data structures in executive images
- Procedure descriptors for universally available routines in executive images and for the embedded system routines
- SYSGEN parameters
- A symbol vector with entries for the data, parameter, routine, and procedure names to be universally available through SYS.EXE

These areas are described in the sections that follow.

32.2.1 Embedded System Routines

Several small routines are defined in SYSTEM_ROUTINES and included in SYS.EXE. Most are routines to which procedure descriptors for executive image routines are initially bound, for example, EXE$LOAD_ERROR and EXE_RSB. If an attempt is made to call an executive image procedure prior to the loading of the image that contains it, control is transferred to one of these routines instead.

Others include the last chance condition handlers for unexpected executive and kernel mode exceptions and the XDELTA-related routines INI$BRK and INI$MASTERWAKE.

32.2.2 Library Procedures

SYS.EXE includes a number of procedures that implement low-level operations like division, character string operations, and field operations. Most calls to these procedures are compiler-generated.

Although a typical image's references to these procedures would be resolved through the Run-Time Library shareable image LIBOTS.EXE, an executive image cannot be linked this way. One alternative is for each executive image to link with an object library containing the procedures the image needs. A simpler and more efficient alternative is to build the executive with only a single copy of each procedure.

Several areas in SYS.EXE are related to these procedures: one for their code, another for read-only data, and another for their linkage section.

32.2.3 Procedure Descriptors for Executive Image Routines

A universally available procedure or routine in an executive image is represented by both a procedure descriptor in the executive image and a bound procedure descriptor in SYS.EXE. A bound procedure descriptor contains the address of a second procedure descriptor to which the first is initially bound. The code entry point specified by the bound procedure descriptor is the address of a minimal routine that calls the second procedure. This provides a mechanism by which the target of a procedure call can be resolved at run time. Section 32.2.6 and Figure 32–2 show an example of this mechanism.

The bound executive image procedure descriptors found in SYS.EXE are defined in module SYSTEM_ROUTINES, usually through its macro DEFINE_ROUTINE and, less commonly, through the macros DEFINE_ROUTINE_INTERRUPT and DEFINE_ROUTINE_EXCEPTION. The *censored* assembler generates a bound procedure descriptor for each invocation of one of these macros. The difference among the macros is the procedure to which the generated procedure descriptor is initially bound: a procedure descriptor defined through DEFINE_ROUTINE is bound to EXE$LOAD_ERROR; one defined through either of the other two macros is bound to EXE$HALT. Example 32–1 shows two such macro invocations. Section 32.8 describes the use of the VERSION_MASK keyword.

Example 32–1 Definition of Executive Image Procedure Descriptors

```
;  Procedure descriptors from SYSTEM_ROUTINES
;
        DEFINE_ROUTINE -
                EXE$ALLOCIRP,-
                VERSION_MASK=<MEMORY_MANAGEMENT>
;
        DEFINE_ROUTINE_EXCEPTION -
                EXE$FRKIPL8DSP,-
                VERSION_MASK=<IO>
```

When an executive image is loaded into system space, its SYS.EXE procedure descriptors are reinitialized to point to the corresponding procedure descriptors in the loaded executive image. If the image is permanently loaded, its SYS.EXE symbol vector entries are also reinitialized to point to addresses in the loaded executive image. (As a result, any references to those procedures that are fixed up

through the updated symbol vector entries do not reference the bound procedure descriptors in SYS.EXE.) If the image is removable, its symbol vector entries are not modified; they continue to point to the bound procedure descriptors and associated transfer routines.

32.2.4 System Data

Module SYSTEM_DATA_CELLS contains data accessed by multiple executive images and by other images that link with SYS.EXE. Data cells accessed by only a limited set of routines typically reside in the same executive image as the routines.

A data cell located in one executive image and referenced by another is represented by a pointer in SYS.EXE, although if the data cell is small, the cell itself resides in SYS.EXE to minimize overhead. If the data cell is represented by a pointer, the pointer is modified when the image is loaded to contain the loaded address of the data. Its symbol name has the type AR to indicate that it contains the address of a record or a structure. For example, LNM$AR_SYSTEM_DIRECTORY contains the address of the logical name table system directory, part of LOGICAL_NAMES.EXE.

The local macro DEFINE_DATA_CELL is invoked for each data cell or structure, along with a VAX MACRO directive to allocate and possibly initialize storage for the cell or structure. Example 32–2 illustrates the use of this macro to define cells within SYS.EXE and to define a pointer to data in another executive image. There is no difference in the macro invocation for these two kinds of definition. Section 32.8 describes the use of the VERSION_MASK keyword.

Example 32–2 Definition of Executive Data Cells

```
; Head of PFN lists
        DEFINE_DATA_CELL -
                PFN$AL_HEAD,-
                VERSION_MASK=<MEMORY_MANAGEMENT>
       .LONG    0,0,0

; Pointer to system logical name table directory
        DEFINE_DATA_CELL -
                LNM$AR_SYSTEM_DIRECTORY,-
                VERSION_MASK=<LOGICAL_NAMES>
        .LONG 0
```

When an executive image is mapped and loaded into system space, its data cell pointers in SYS.EXE are reinitialized to point to their corresponding data structures in the loaded executive image.

Section 32.4.1 describes other criteria for choosing whether to locate a system data cell or structure in SYS.EXE or in a particular executive image.

32.2.5 SYSGEN Parameters

The SYSGEN parameters area is defined in module SYSPARAM. This area contains all the SYSGEN parameters. For coordination with SYSBOOT, which copies the current parameters to this area during system boot, all SYSGEN parameters are virtually contiguous. Chapter *Operating System Initialization and Shutdown* describes this in more detail.

SYSGEN parameters are part of SYS.EXE rather than of a particular executive image so that they can be referenced directly. No one executive image references them most often; they are widely referenced from many executive images and from other images linked with SYS.EXE.

32.2.6 Symbol Vector

A symbol in a shareable image that is to be externally available to other images, such as the name of a data structure or a procedure, must be a universal symbol. A universal symbol must have been defined as a global symbol when the module that contains it was compiled, and it must be declared in a SYMBOL_VECTOR option statement when the image that contains it is linked.

To describe the universal symbols, the linker creates two structures in the shareable image file: a global symbol table and an image section called a symbol vector. Each global symbol has an entry in the global symbol table containing the name of the symbol and the offset of the symbol's entry in the symbol vector. The global symbol table is not loaded into memory with the rest of the image. It is used only during linking as an index into the symbol vector. The symbol vector is loaded into memory as part of the image and is used during image activation and fixup. Each symbol vector entry contains the offset in the image of the location that the symbol names.

SYS.EXE's global symbol table and symbol vector have entries for all the externally visible executive routines and data that make up the VMS executive. Some of the routines and data are within SYS.EXE itself, for example, the library routines, described in Section 32.2.2, and SYSGEN parameters. Many entries in SYS.EXE's symbol vector, however, represent global symbols in executive images. Example 32–3 shows the definition of a number of SYS.EXE symbol vector entries; definitions of some of these universal symbols are shown in the previous examples.

Each such symbol is defined as a global symbol at compile time in one of the modules that comprise SYS.EXE and is declared in SYS.EXE's SYMBOL_VECTOR link option. The symbol vector entry points to the SYS.EXE location named by the symbol. Each such symbol is also defined as a global symbol in an executive image. The executive image is linked with a VECTOR_TABLE option statement that identifies SYS.EXE as containing its symbol vector. After processing the modules that make up an executive image, the linker searches the specified symbol vector. For each symbol in the symbol vector that is defined as a global symbol in a module of the executive image, the linker creates a vectored universal symbol in the executive image's global symbol table. A vectored universal symbol has two values: the address of the location it names in the executive image and the offset of its entry in the SYS.EXE symbol vector.

Example 32–3 Definition of SYS.EXE Universal Symbols

```
! Following are excerpts from [SYS.COM]SYSLNK.OPT, the option file
! for linking SYS.EXE
!
        . . .
SYMBOL_VECTOR = (-                              !Vectors from SYSPARAM
        SGN$GL_NPAGEDYN         = DATA, -
        SGN$GL_NPAGEVIR         = DATA, -
        SGN$GL_PAGEDYN          = DATA, -
        . . .
        SYS$GB_NET_CALLOUTS     = DATA)
!
SYMBOL_VECTOR = (-                              !Vectors from SYSTEM_ROUTINES
        ACP$ACCESS              = PROCEDURE, -
        . . .
        EXE$ALLOCIRP            = PROCEDURE, -
        EXE$FRKIPL8DSP          = PROCEDURE, -
        . . .
        EXE$SAVE_ALIGN_FAULT_DATA = PROCEDURE)
!
SYMBOL_VECTOR = (-                              !Vectors from SYSTEM_DATA_CELLS
        . . .
        PFN$AL_HEAD             = DATA, - !Head of PFN lists
        PFN$AL_TAIL             = DATA, - !Tail of PFN lists
        . . .
        EXE$GQ_DELETED_KFE      = DATA)   !Deleted known file entry list
```

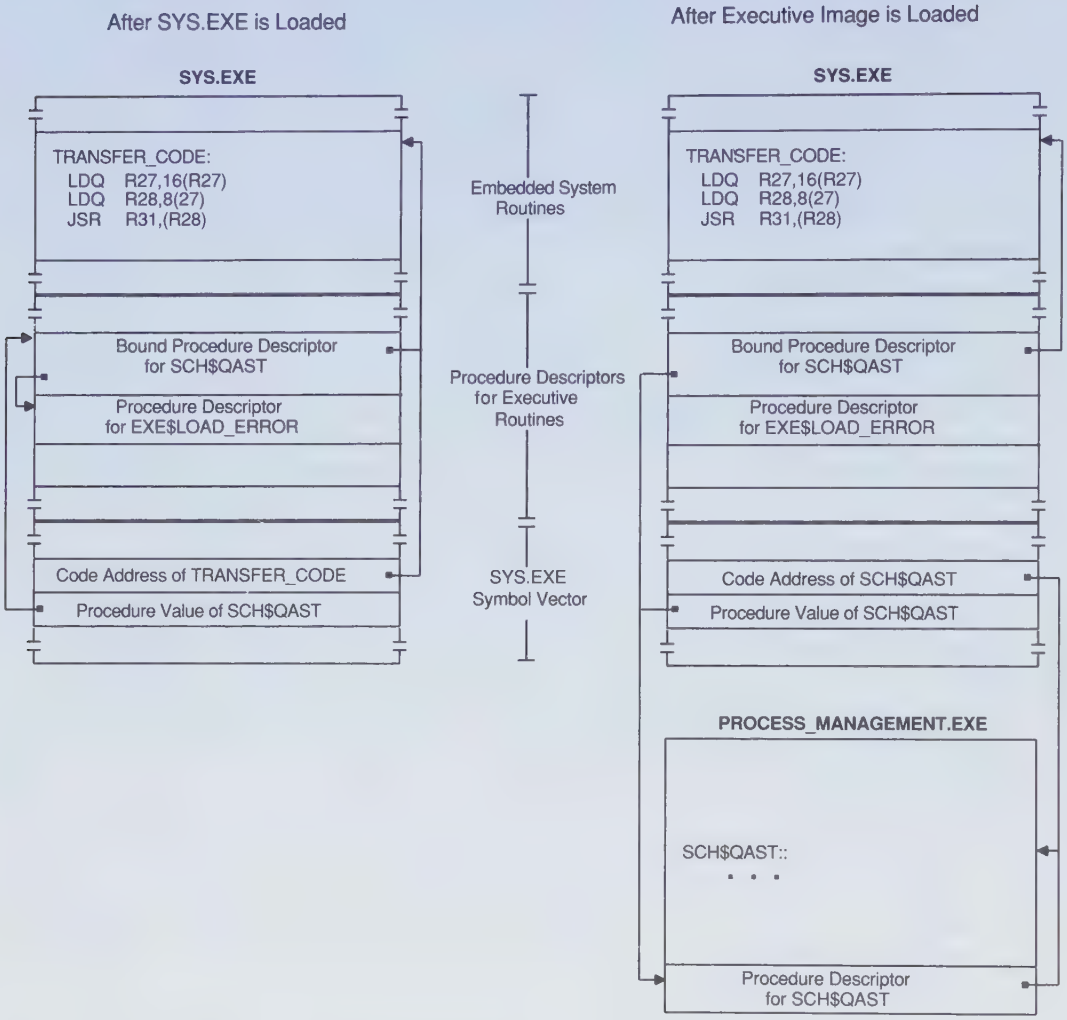
When a nonexecutive image is linked with SYS.EXE, the linker searches SYS.EXE's global symbol table for each universal SYS.EXE symbol referenced to get the offset of its associated symbol vector entry. At link time, SYS.EXE's symbol vector entries contain the following:

- Relative addresses of data cells within it
- Relative addresses of cells that will eventually point to data cells in executive images
- Relative code entry and procedure descriptor addresses, both those that will eventually point to routines in executive images and those that will point to routines in SYS.EXE

When SYSBOOT loads SYS.EXE into memory, it relocates the addresses in the loaded symbol vector entries to reflect the address at which SYS.EXE was loaded. It also records the beginning and end addresses of the symbol vector within the loaded SYS.EXE in the cells EXE$GL_SYS_SYMVEC and EXE$GL_SYS_SYMVEC_END.

As each executive image is loaded, the symbol vector entries in SYS.EXE that should point to data cells in that image are updated. Whether the image is permanent or removable, its SYS.EXE procedure descriptors are rebound to the actual procedures in the loaded executive image. If the image is permanent, the symbol vector entries in SYS.EXE that had pointed to the bound SYS.EXE procedure descriptors are updated to point directly to the procedures in the loaded image. If the image is removable, the symbol vector entries are left pointing to the bound SYS.EXE procedure descriptors to facilitate removal. This change happens as part of executive image loading. Figure 32–2 illustrates the changes in an example SYS.EXE symbol vector entry and bound procedure descriptor.

Figure 32–2 SYS.EXE Symbol Vector Entry and Bound Procedure Descriptor



When an image linked with SYS.EXE is activated, its references to SYS.EXE symbols are fixed up with the current contents of the corresponding symbol vector entries in the loaded SYS.EXE.

32.3 The Base Image SYS$PUBLIC_VECTORS.EXE

SYS$PUBLIC_VECTORS.EXE contains a symbol vector whose entries represent all the universally available names for system services provided by executive images. An image requests a particular system service by calling SYS$*service_name*, the universally available system service name. The linker resolves system service names using global definitions from SYS$LIBRARY:SYS$PUBLIC_VECTORS.EXE, which it searches by default.

VMS guarantees that the offsets of entries in SYS$PUBLIC_VECTORS.EXE's symbol vector will remain the same in subsequent versions of [*censored*] VMS. This ensures that an image linked with SYS$PUBLIC_VECTORS.EXE can execute on a later version than the one with which it linked.

The base image cells EXE$GL_PUBLIC_VECTOR_SYMVEC and EXE$GL_PUBLIC_VECTOR_SYMVEC_END contain the beginning and end addresses of the symbol vector within the loaded SYS$PUBLIC_VECTORS.EXE.

Built from the module SYSTEM_SERVICES, SYS$PUBLIC_VECTORS.EXE contains procedure descriptors for system service transfer routines and the system service transfer routines themselves. It also contains several universally available data cells holding system addresses that must be accessible to images that do not link with SYS.EXE. An example is the address of SYS$CALL_HANDL's procedure descriptor. Chapter *Condition Handling* describes this procedure and its purpose.

A system service transfer routine is a minimal procedure that executes in the mode of the caller and that serves as a bridge to the actual procedure implementing the service request. The actual procedures are within executive images and typically execute in an inner access mode.

When an executive image is loaded, each system service in it that executes in an inner access mode is assigned a unique change mode number. Its system service transfer routine is overwritten to contain an instruction that loads this change mode number into R0 and executes a `CALL_PAL CHMx` instruction.

When an image requesting a system service calls its transfer routine, the transfer routine executes these instructions, resulting in an exception. The change mode exception service routine uses the change mode number to index a table that contains the addresses of the actual system service procedure descriptor and its code entry point. It then calls the actual procedure.

When an executive image is loaded that contains a mode of caller system service, the contents of the SYS$PUBLIC_VECTORS.EXE symbol vector entry for that universal system service name are updated to point to the executive image. As part of executive image initialization, described in Section 32.7.5, the procedure descriptor in SYS$PUBLIC_VECTORS.EXE is bound to the procedure in the executive image.

Chapter *System Service Dispatching* contains further information about system service transfer routines, change mode dispatching, and synchronous system services. Section 32.7.5 describes the initialization of system service transfer routines.

32.4 Executive Images

The two base images contain relatively little executable code. Their executive image procedure descriptors generally cause dispatch to routines in loaded executive images.

Each executive image consists of data, routines, and initialization code specific to the image's functions and features. In most cases, to simplify maintenance and enhancement, routines supporting related functions and features are collected into the same image. In some cases, routines used early in system initialization are combined into an executive image, such as EXEC_INIT.EXE. Chapter *Bootstrap Processing* lists the executive images and summarizes their contents.

[*Censored*] VMS differs from VAX/VMS in that all images loaded into system space other than the two base images are executive images. That is, device drivers and images containing, for example, CPU-specific support, are implemented as executive images.

Although maintainability considerations might lead to more specialized executive images and thus more executive images, several [*censored*] hardware and software considerations make decreasing the number of executive images desirable:

- The larger physical page size increases the average unused physical memory for an image section that is not an integral number of pages.
- Branch destination hint optimizations can be effective only within an image. With the branch destination hint mechanism the CPU uses the otherwise unused displacement bits in a JSR or JMP instruction to calculate the low-order bits of the destination's address. The CPU performs an early instruction cache lookup of the likely destination address. If the hint bits are correct, the instruction at the destination address can be fetched more quickly. VMS software only calculates branch destination hints when the branch and its target are in the same image.
- Calls within an image can be made more efficiently than calls between two images. [*Censored*] compilers generate object language records that enable the linker to replace intra-image call sequences with BSR instructions.

32.4.1 Data in an Executive Image

A data cell private to routines in an executive image resides in that image. A data cell accessed by routines in multiple executive images can be placed in SYS.EXE or in one of the executive images. If the data itself is not in SYS.EXE, SYS.EXE contains a pointer to it for use by the other executive images. That is, a routine in one executive image does not directly reference data in another but instead makes an indirect reference through a SYS.EXE pointer.

A data structure whose size is likely to vary from version to version is traditionally stored in an executive image. A cell in SYS.EXE points to the structure if it is referenced from other executive images.

Certain data cells reside in executive images even though they are small and unlikely to change size. A data cell that is referenced primarily by routines within the image is typically in the image itself, to reduce the access overhead for the most frequent references.

Writable data cells referenced by commonly executed code paths are stored in the image with the most time-critical accesses.

32.4.2 Structure of a Linked Executive Image

An executive image is implemented as a form of shareable image. Like any shareable image, it has a global symbol table, image section descriptors, and an image activator fixup section. Unlike other types of shareable image, an executive image does not contain a symbol vector. Instead, its universally available procedures and data cells are reached through entries in the loaded SYS.EXE and SYS$PUBLIC_VECTORS.EXE symbol vectors.

The internal structure of an executive image is more constrained than that of a typical shareable image. An executive image is allowed at most one image section of each of the following types and no others:

- Nonpageable read-only, for code
- Nonpageable writable, for both read-only and writable data, locations containing addresses that must be relocated at image activation, and the linkage section for nonpageable code
- Pageable read-only, for code
- Pageable writable, for both read-only and writable data, locations containing addresses that must be relocated at image activation, and the linkage section for pageable code
- Initialization section, for initialization procedures and their associated data and linkage section
- Image activator fixup section

The first five of these image sections are defined as program sections (PSECTs) within the modules of an executive image. An executive image is linked so as to place the initialization routine section after the others. The linker creates the image activator fixup section as the last image section. The initialization routine and image activator fixup section are last because their address space is deallocated after initialization and fixups are done.

The image activator fixup section (see Chapter *Image Activation and Exit*) contains information needed to relocate address references from the loaded image to locations within itself and to fixup references to locations in the base images. These references are found in procedure descriptors and linkage pairs and also generated by the VAX MACRO directives .ADDRESS and .ASCID and their high-level language equivalents.

The first four sections allow for the combinations of pageability and protection required for executive code and data. Because they have different virtual memory characteristics, each must begin at a page boundary. (Protection and pageability requirements could be met by placing read-only data with the code in the appropriate image section, as they are in a VAX/VMS executive image. However, a *censored* processor typically has both a data translation buffer and an instruction translation buffer, as described in Chapter *Memory Management Overview*. Mixing data and code in the same page could result in two translation buffer entries mapping the same physical page, an inefficient use of a relatively scarce resource that could reduce overall performance.) That each section begins on a page boundary can result in unused disk and address space at the end of each image section. An executive image on disk may have on average half a disk block

unused at the end of each image section. Section 32.4.3 discusses how the image sections of an executive image are mapped into virtual address space and what unused space might result.

Constraining the number of image sections may limit the potential unused space; it also simplifies the loading mechanism.

Most modules invoke the DECLARE_PSECT macro to define standard executive PSECT names and attributes. Each image is built with a linker options file that collects and orders the image sections. Table 32–1 lists the clusters and PSECTs that make up a typical executive image. It shows some of the modules that make contributions to the PSECTs. This information is extracted from the image map of SYS$VM.EXE.

Table 32–1 Organization of SYS$VM.EXE, a Typical Executive Image

PSECT Name	Object Module Name
NONPAGED_READONLY_PSECTS Cluster	
EXEC$HI_USE_PAGEABLE_CODE[1]	SYSCREDEL
	. . .
EXEC$NONPAGED_CODE	EXECUTE_FAULT
	. . .
NONPAGED_READWRITE_PSECTS Cluster	
EXEC$HI_USE_PAGEABLE_DATA[1]	SYSCREDEL
	. . .
EXEC$HI_USE_PAGEABLE_LINKAGE[1]	SYSCREDEL
	. . .
EXEC$NONPAGED_DATA	OSWPSCHED
	. . .
EXEC$NONPAGED_LINKAGE	EXECUTE_FAULT
	. . .
PAGED_READONLY_PSECTS Cluster	
EXEC$PAGED_CODE	GSD_ROUTINES
	. . .
PAGED_READWRITE_PSECTS Cluster	
EXEC$PAGED_DATA	GSD_ROUTINES
	. . .
EXEC$PAGED_LINKAGE	GSD_ROUTINES
	. . .

[1]This PSECT represents code or data that could be paged but that has been placed in a nonpageable image section to improve performance.

Table 32–1 (Cont.) Organization of SYS$VM.EXE, a Typical Executive Image

PSECT Name	Object Module Name
INITIALIZATION_PSECTS Cluster	
EXEC$INIT_000	SYS$DOINIT
EXEC$INIT_001	SYS$DOINIT
	. . .
EXEC$INIT_002	SYS$DOINIT
EXEC$INIT_CODE	SYS$DOINIT
	LOCK_SYSTEM_PAGES
	. . .
EXEC$INIT_LINKAGE	SYS$DOINIT
	LOCK_SYSTEM_PAGES
	. . .
EXEC$INIT_SSTBL_000	SYS$DOINIT
EXEC$INIT_SSTBL_001	SYS$DOINIT
	SYSADJSTK
	. . .
EXEC$INIT_SSTBL_002	SYS$DOINIT
Image Activator Fixup Section	
	(Built by the linker)

An executive image is further constrained in that it must be linked with the /NATIVE_ONLY qualifier and must not call routines in translated images. To be loaded sliced (see Section 32.4.3), the image must also be linked with the qualifier /SECTION_BINDING=(CODE,DATA) and contain no relative references from one image section to another.

[*Censored*] VMS supports executive images through several shareable image mechanisms, some of them unique to executive images:

- The linker SYMBOL_VECTOR option identifies the symbols in a shareable image that are to be universal. For a VMS executive image, the symbol vector entries are in SYS.EXE rather than in each executive image.
- A vectored universal symbol has two values: the relative address of the symbol in the executive image, and the offset of the symbol's entry in the SYS.EXE symbol vector.
- The image header contains space for an array of version numbers, described in Section 32.8.

- The linker COLLECT qualifier /ATTRIBUTES identifies special characteristics of a particular image section. The possible values for the qualifier are RESIDENT, to designate a nonpageable image section, and INITIALIZATION_CODE, to designate the initialization image section. These values initialize the image section descriptor flags EISD$V_RESIDENT and EISD$V_INITIALCODE.

32.4.3 Mapping an Executive Image

An executive image can be mapped into system space in one of two ways:

- In contiguous virtual pages with the virtual layout created by the linker
- With some of its sections in granularity hint regions and thus in discontiguous virtual addresses

By default most executive images are mapped with sections in granularity hint regions. This option provides better performance.

At the [*censored*] hardware level, virtual address translation is based upon the translation buffer (TB). A TB caches page table entries (PTEs) for recently referenced virtual addresses. When a virtual address to be translated is not in the TB (a TB miss), the hardware generates a trap to a privileged architecture library (PALcode) routine. The PALcode routine fetches the correct PTE from memory and loads it into an available TB entry. If all TB entries are in use, one must be reused. Translating an address when its PTE is already in the TB (a TB hit) is significantly faster than taking a TB miss as part of the translation.

A [*censored*] TB optionally supports a feature called a granularity hint, by means of which one TB entry can represent a group of physically and virtually contiguous pages with identical PTE characteristics (protection, validity, and fault enablings). Use of granularity hints improves performance by increasing the number of apparent TB entries and thus reducing TB misses. The number of pages in a group described by one TB entry can be 8, 64, or 512 pages and is specified in the PTE of each page in the group. The details of a particular TB, such as how many entries support granularity hints and group size, vary with CPU type.

A group of pages that can be described by one TB entry is called a granularity hint region. A granularity hint region must be on a naturally aligned boundary. For example, a granularity hint region of 64 pages must be on a physical and virtual 64-page boundary. For VMS to make use of granularity hints, it must reserve blocks of physical memory and virtual address space early in system initialization to ensure that the address constraints can be met.

By default it allocates one or more granularity hint regions for each of the following purposes:

- Base and executive images' nonpaged code
- Base and executive images' nonpaged read-only and writable data
- Nonpaged dynamically allocated system data, such as the PFN database

The granularity hint region or regions associated with each of these uses is commonly referred to as a huge page. Section 32.5.2 discusses the data structures that describe huge pages.

VMS allocates address space for huge pages at the low end of system space. Because all pages in a granularity hint region must be in physically contiguous pages, VMS does not page the virtual pages occupying a huge page; they must all be resident. If the SYSGEN parameter S0_PAGING is nonzero to disable paging of executive images, their usually pageable image sections occupy the same huge pages as the nonpageable sections.

If any shareable images are installed permanently resident, their code also occupies the first huge page (see Chapter *Memory Management System Services*).

The effect of loading an executive image's nonpageable image sections into huge pages is that the image does not retain the virtual layout created by the linker. The nonpaged read-only section of the image (which contains only code) is loaded into one huge page; its nonpaged writable section is loaded into another; and its pageable sections are in yet a third part of system virtual address space. When resident, the image's pageable sections occupy ordinary physical pages.

The memory represented by a huge page is allocated to an image section in a unit called a slice. Different huge pages can have different slice sizes. VMS picks the size of the code page slice so as not to reduce the effectiveness of branch destination hint optimizations. On current systems the code slice size is 8K bytes. VMS specifies the size of the data page slice as 512 bytes to guarantee that a data section loaded into a huge page retains requested alignment up to a 512-byte boundary.

An integral number of slices is allocated for each image section. For example, the nonpageable read-only section of one executive image occupies one or more contiguous slices that are immediately adjacent to those of the previous and next images. An image mapped into a huge page is said to be sliced.

Figure 32–3 shows an example extract from the first two huge pages of a system whose executive images have been loaded sliced. The code page slice size is 8K bytes, and the data page slice size is 512 bytes.

Bit 1 (SGN$V_EXEC_SLICING) of the SYSGEN parameter LOAD_SYS_IMAGES affects whether VMS loads base and executive images sliced. Its default value is 1, enabling slicing of executive images. If its value is 0, no executive images are loaded sliced. A nonzero value in the SYSGEN parameter SHRIMG_MAXMEM enables the loading of installed resident images into the code page. Even if both values are zero, a huge page is still created for nonpageable dynamically allocated system data.

Figure 32–3 Executive Images Loaded Sliced

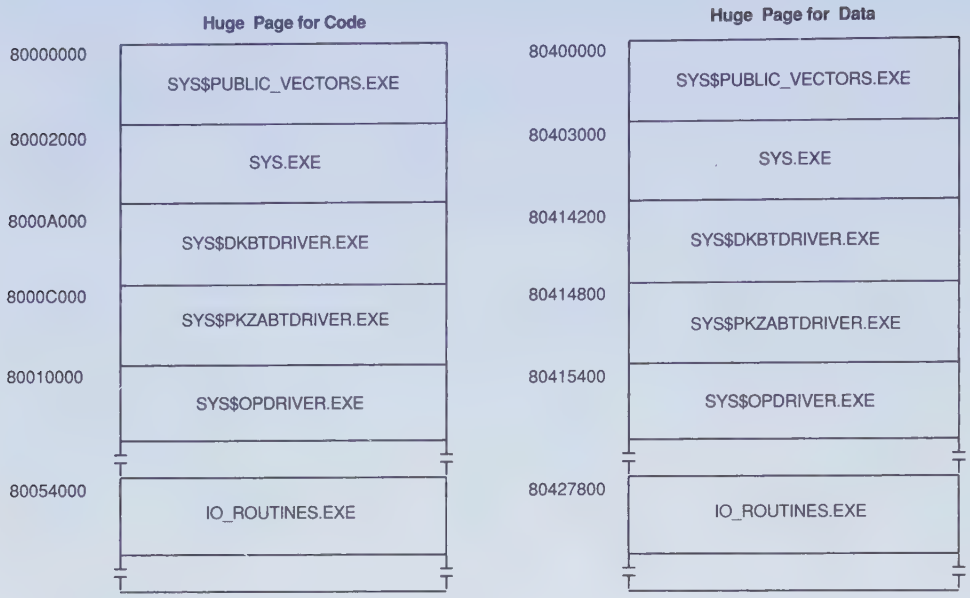
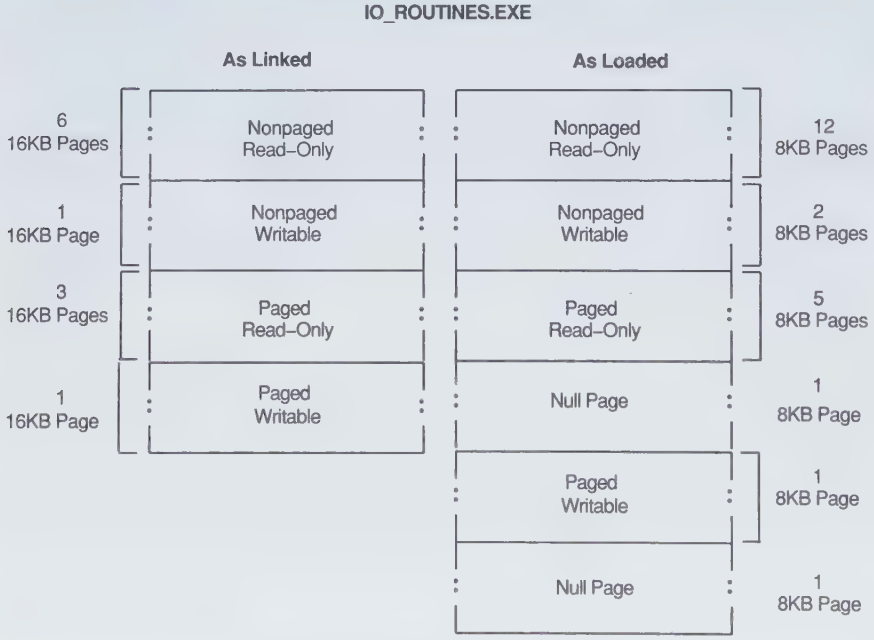


Figure 32–4 An Executive Image Loaded Nonsliced



It is possible to specify that a particular executive image not be loaded into a huge page, even if slicing is enabled, through a flag passed as an argument to the procedure that loads executive images. VMS sets this flag, for example, so that the image EXEC_INIT.EXE, which is deallocated at the end of system initialization, is not loaded into a huge page.

Loading the different sections of an executive image into different virtual addresses adds some complexity to relocating absolute addresses and fixing up references to external addresses. It also requires that the data structure that describes a loaded executive image specify the location and length of each image section (see Section 32.5.1).

When an executive image is not mapped into a huge page, a different issue can affect its mapping. The *censored* architecture allows for several different page sizes. Any particular CPU type has a given page size of 8K, 16K, 32K, or 64K bytes. To run on a CPU with a particular page size, an image must have been linked to that page size or a larger one. (An image linked to a smaller page size might have an image section that must begin at an address that is a multiple of the smaller page size but not of the larger.)

When an image is loaded on a CPU with a smaller page size than the one to which the image was linked, each of the image sections can begin on a page boundary and be mapped with appropriate pageability and protection. However, if an image section ends with unused address space that is one or more of the smaller pages in length, the unused space is mapped with one or more null pages, that is, no-access PTEs. Figure 32–4 shows an executive image linked to a 16K byte page size and mapped on a CPU with an 8K byte page size.

32.4.4 Symbol Resolution and Executive Images

In general, a *censored* image can refer to symbols and addresses within itself and in shareable images with which it is linked. An executive image is more constrained in that it may only be linked with the base images SYS.EXE and SYS$PUBLIC_VECTORS.EXE and thus can only refer to its own symbols or symbols in one of the base images. Some symbolic references are self-relative and can be fully resolved at compile time. Most, however, are absolute and cannot be resolved until address space is assigned. Absolute references in an executive image cannot be resolved before address space is assigned to the image and the base images.

The base images are loaded first, followed by executive images. As each image is loaded, address references are resolved:

- Absolute references from an image to locations within itself are relocated.

  These are found in procedure descriptors and linkage pairs and also generated by VAX MACRO directives such as .ADDRESS and .ASCID or by their high-level language equivalents. LDR$RELOCATE_IMAGE, in module LDR_RELOCATE_FIXUP, relocates such address references in a base or executive image. Chapter *Image Activation and Exit* gives a more detailed description of address relocations.

- References from an executive image to universal symbols defined in SYS.EXE or SYS$PUBLIC_VECTORS.EXE are fixed up.

These references are found in procedure descriptors and linkage pairs and also generated by VAX MACRO directives such as .ADDRESS and .ASCID or by their high-level language equivalents. They derive from system service requests, accesses to systemwide data or data pointers in SYS.EXE, and invocations of routines defined in SYS.EXE's symbol vector.

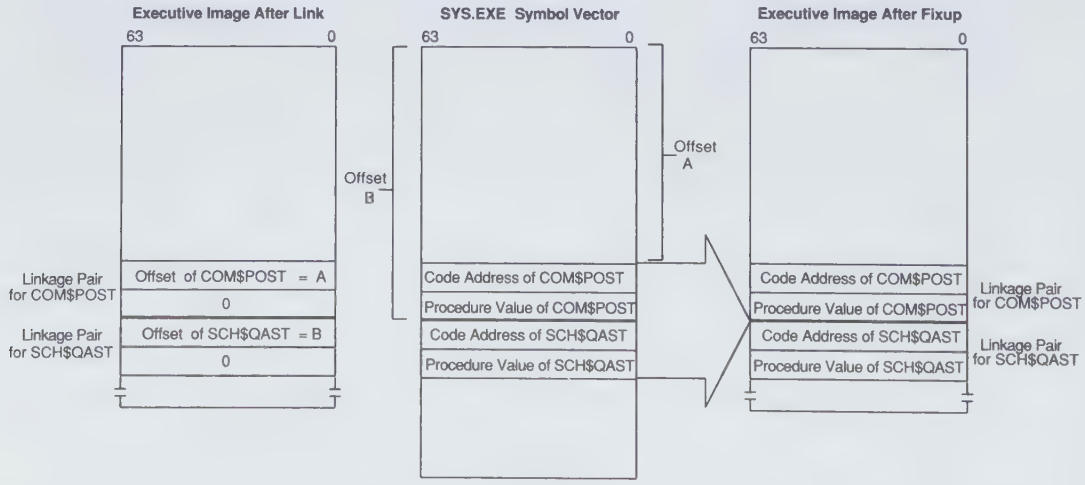
In a fixup, one or two addresses from a previously relocated symbol vector entry replace a reference in an image. LDR$FIXUP_IMAGE, in module LDR_RELOCATE_FIXUP, fixes up address references in executive images using the already relocated contents of SYS.EXE and SYS$PUBLIC_VECTORS.EXE symbol vector entries. Chapter *Image Activation and Exit* gives a more detailed description of fixups.

As described in Section 32.2.3, one executive image calls a routine in another executive image through a SYS.EXE universal symbol. The relocated value of the universal symbol is initially a procedure descriptor in SYS.EXE bound to an error-handling routine. If image A, loaded first, calls routine Y in image B, which is not yet loaded, then A's reference to Y is fixed up as the address of the relocated bound procedure descriptor. The relocated procedure descriptor is still bound to an error routine.

When image B is loaded, the procedure descriptor is rebound to routine Y in image B. If code in image A then called routine Y, control would be transferred to the SYS.EXE procedure, which transfers control to routine Y. To avoid the indirection, after all executive images are loaded, "second pass" fixups are performed so that interimage calls can be direct regardless of the order in which the executive images were loaded.

Figure 32–5 shows a simple example of one executive image referencing the external routines SCH$QAST and COM$POST.

Figure 32–5 Symbol Resolution Through SYS.EXE



Both routines are defined in SYS.EXE's symbol vector. After the example executive image is linked, its linkage pairs contain the symbol vector offsets of these symbols. After the executive images that define SCH$QAST and COM$POST are

loaded, SYS.EXE's symbol vector entries are updated with actual addresses of the procedure descriptors and their code entry points. After the example executive image is loaded, its linkage pairs are fixed up to contain the contents of the loaded symbol vector entries.

32.5 Executive Image Data Structures

This section describes the data structures related to supporting the modular executive.

32.5.1 Loadable Image Data Structures

Each loaded base and executive image is described by a nonpaged pool data structure called a loader image data block (LDRIMG).

The LDRIMGs are linked together through LDRIMG$L_FLINK and LDRIMG$L_BLINK in a doubly linked list whose head is at LDR$GQ_IMAGE_LIST. The list is searched before an executive image is loaded to ensure that the image has not already been loaded. XDELTA, DELTA, and the System Dump Analyzer scan the list to determine where executive images have been loaded. Figure 32–6 shows the layout of an LDRIMG. Access to the LDRIMG list is synchronized through the base image mutex, EXE$GQ_BASIMGMTX.

LDRIMG$W_SIZE contains the size in bytes of the structure. LDRIMG$B_TYPE contains the constant DYN$C_LOADCODE. LDRIMG$B_IMGNAMLEN is the start of a counted ASCII string containing the name of the executive image.

LDRIMG$L_FLAGS describes the state of the loaded image and the manner in which it was loaded. Figure 32–6 lists the flag names, omitting the prefix LDRIMG$V_, and their meanings.

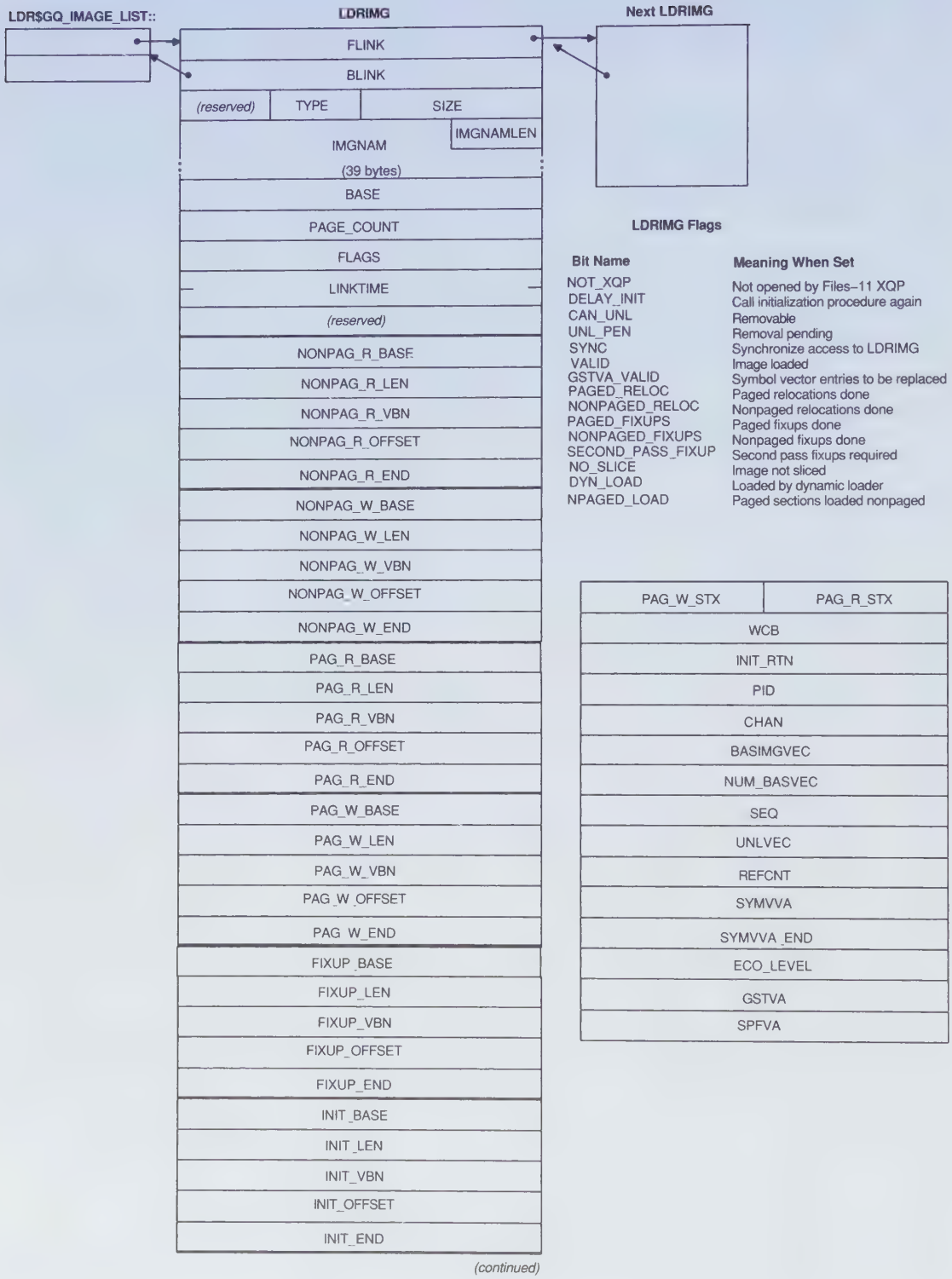
For an image whose sections are loaded into contiguous address space, LDRIMG$L_BASE contains its starting virtual address; for an image loaded sliced, it contains zero. LDRIMG$L_PAGE_COUNT represents the length in pages from the start of the loaded image to the end of its highest image section. The page size is that of the CPU on which the image is loaded.

In LDRIMG$Q_LINKTIME is the system time at which the image was linked. LDRIMG$L_ECO_LEVEL contains revision data. Both fields are copied from the image's image header.

There are five LDRIMG fields for each of the six types of image section. Each field has a unique name, as shown in Figure 32–6. Additionally, the following symbols represent offsets from the beginning of the set of fields for each image section:

- LDRIMG$L_ISD_BASE—The base virtual address at which this image section has been loaded
- LDRIMG$L_ISD_LEN—The size in bytes of this image section
- LDRIMG$L_ISD_VBN—The virtual block number (VBN) in the image file at which this section begins

Figure 32–6 Layout of a Loader Image Data Block



(continued)

- LDRIMG$L_ISD_OFFSET—The virtual offset in the image at which this section begins
- LDRIMG$L_ISD_END—The virtual offset in the image at which this section ends

If an executive image lacks a particular type of image section, most of the fields that represent that section contain 0. After the fixup and initialization sections are deallocated, their fields are cleared. (The LDRIMG$L_ISD_OFFSET and LDRIMG$L_ISD_LEN fields of an image loaded sliced are initialized to –1 and do not change.)

LDRIMG$W_PAG_W_STX and LDRIMG$W_PAG_R_STX contain the global section table indexes of the paged sections, if they exist.

LDRIMG$L_WCB contains the address of the window control block (WCB) that describes the image file on disk.

If the image defines an initialization procedure, LDRIMG$L_INIT_RTN contains the loaded virtual address of its procedure descriptor. After the initialization procedure is done and the initialization section deallocated, this field is cleared.

While an image is being loaded by LDR$LOAD_IMAGE, in module SYSLDR_DYN, LDRIMG$L_PID and LDRIMG$L_CHAN describe the process ID in whose context the image file has been opened and the channel on which it has been opened. After loading is complete, these two fields are cleared.

LDRIMG$L_BASIMGVEC points to a nonpaged pool structure that contains information necessary to restore SYS.EXE symbol vector entries to their original contents after having been modified to point into this executive image. LDRIMG$L_NUM_BASVEC contains the number of such entries in the structure. These fields are meaningful only for a removable image or one loaded dynamically.

LDRIMG$L_SEQ contains a sequence number taken from the contents of EXE$GL_LDR_SEQ at the time the image was loaded. The sequence number uniquely identifies the combination of an executive image and a process attempting to load or unload it. The combination of this field, LDRIMG$L_REFCNT, and the flags LDRIMG$V_SYNC, LDRIMG$V_VALID, and LDRIMG$V_UNL_PEN enables VMS to synchronize attempts to load and unload the same executive image with each other and with concurrent accesses to the image. XDELTA can identify a particular loaded executive image through its sequence number.

If the image defines one or more procedures to be called when the image is unloaded, LDRIMG$L_UNLVEC contains the address of a table within the image that lists those procedures.

For an image that can be removed from memory, LDRIMG$L_REFCNT counts the number of current references made to the image. When the reference count is nonzero, the image may not be unloaded.

LDRIMG$L_SYMVVA and LDRIMG$L_SYMVVA_END delimit a symbol vector within a loaded base image.

LDRIMG$L_GSTVA, when flag LDRIMG$V_GSTVA_VALID is set, points to a nonpaged pool structure containing entries from the image's global symbol table that cannot be fixed up until paging is possible. After the entries are processed, the pool is deallocated, and both the flag and field are cleared.

LDRIMG$L_SPFVA, when flag LDRIMG$V_SECOND_PASS_FIXUP is set, points to a nonpaged pool structure containing entries that represent external fixups to SYS.EXE procedures that need to be redone after the executive image containing the target procedure is loaded. After the entries are processed, the pool is deallocated, and both the flag and field are cleared.

32.5.2 Data Structures to Describe Huge Pages

Huge pages are allocated during system initialization prior to the loading of executive images. Section 32.4.3 describes how executive images are loaded sliced into huge pages.

SYSBOOT calls LDR$INIT_MEM, in module LDR_MEM_INIT, which allocates the physical pages and system page table entries (SPTEs) for huge pages. By default it allocates three huge pages. (Actually, the second huge page is currently allocated as two granularity hint regions, each 64 pages long. This makes it possible to deallocate the second piece if it is unused.)

LDR$INIT_MEM also allocates a page of memory to build data structures to describe the huge pages; for each huge page it builds a loader huge page descriptor (LDRHP) and initializes a bitmap to reflect allocations within the huge page.

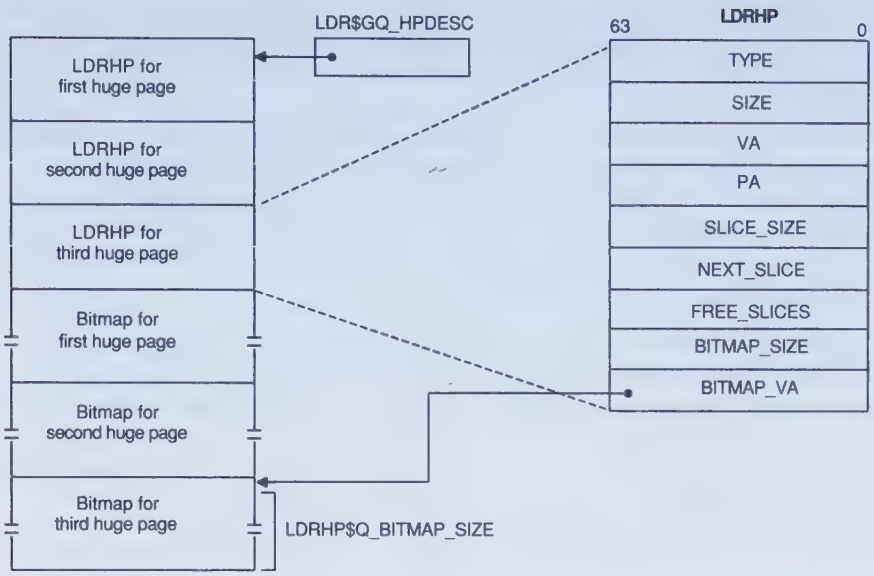
In a later state of initialization the structures and bitmaps are copied to a piece of nonpaged pool whose address is recorded in LDR$GQ_HPDESC. Once SYSINIT begins to execute, access to these structures and bitmaps is synchronized with the base image mutex, EXE$GQ_BASIMGMTX. Figure 32–7 shows the layout of these structures.

LDRHP$Q_TYPE identifies the type of huge page: read-only image sections, writable image sections, or systemwide writable data. LDRHP$Q_SIZE contains the size of the huge page in bytes; LDRHP$Q_PA, its starting physical address; and LDRHP$Q_VA, its starting virtual address.

LDRHP$Q_SLICE_SIZE contains the granularity of allocation from this huge page. On current [*censored*] implementations, the granularity of allocation is 8K bytes for a code (read-only image section) page and 512 bytes for either of the other pages. LDRHP$Q_FREE_SLICES contains the number of available slices left in the huge page. The bitmap, whose starting virtual address is in LDRHP$Q_BITMAP_VA, has one bit per slice. If the value of the bit is 1, the slice is available; if 0, it has been allocated. The size in bytes of the bitmap is in LDRHP$Q_BITMAP_SIZE. LDRHP$Q_NEXT_SLICE contains the number of the first free slice. (For simplicity, the figure shows the bitmaps beginning on 64-byte boundaries. However, the second and third bitmaps can begin on arbitrary byte boundaries.)

The routines that allocate and deallocate space in a huge page are LDR$ALLOC_SLICE and LDR$DEALLOC_SLICE, in module LDR_MEM_ALLOC.

Figure 32–7 Huge Page Data Structures



32.6 Executive Image Loading

Executive images are loaded and initialized at several stages in system initialization. An image is initially loaded at one particular stage. However, if the image contains pageable sections and is loaded before paging is possible, relocations and fixups in its pageable sections must be deferred to a later stage of system initialization. An image's initialization procedure can potentially be executed when the image is loaded and again at any of several succeeding stages of initialization. In general, loading of executive images is deferred to the later stages of system initialization, if possible, for simplicity.

The major stages of system initialization at which images are loaded, initialized, or fixed up are as follows:

1. SYSBOOT.EXE, the secondary bootstrap program, which initializes system space and loads the base images and executive images needed during the early stages of system initialization
2. EXEC_INIT.EXE (routine EXE$INIT), the executive image that performs initialization of system data and that loads most of the other executive images
3. The swapper process, the first process to execute
4. The SYSINIT process, which loads a few executive images
5. The startup process, in whose context the routine LDR$RELEASE_MEM, in module LDR_MEM_ALLOC, is invoked to complete fixup of executive images

Chapter *Bootstrap Processing* lists the images loaded at each stage of system initialization. That chapter and Chapter *Operating System Initialization and Shutdown* describe these and other stages of system initialization in detail. This section is concerned only with their role in the loading and initialization of executive images.

The stages of system initialization are divided into substages, primarily to control loading and initialization of executive images. The system global EXE$GL_STATE describes these substages with a bit set to represent each substage that has been reached. The macro $BOOSTATEDEF defines symbolic values for these bits. Table 32–2 lists the symbols in the order in which their bits are set. For simplicity, it omits the prefix BOOSTATE$V_.

Executive images are loaded during system initialization by two versions of LDR$LOAD_IMAGE. The first, in module SYSLDR, which is sometimes referred to as the boot loader, is linked with SYSBOOT.EXE and EXEC_INIT.EXE. The other version, in module SYSLDR_DYN, which is sometimes referred to as the dynamic loader, is part of the executive image SYSLDR_DYN.EXE. Routines used by both versions of LDR$LOAD_IMAGE are in module SYSLDR_COMMON.

The loading and initialization of executive images are described in the sections that follow.

Table 32–2 Boot Stages and Conditions (EXE$GL_STATE Bits)

Bit Name	Set By	Meaning
SYSBOOT	SYSBOOT	SYSBOOT has begun
EXEC_SLICING[1]	LDR$INIT_MEM	Executive images should be sliced
POOL_INIT	SYSBOOT	Nonpaged pool allocation is possible
INIT	EXE$INIT	EXE$INIT has begun
CONSOLE	EXE$INIT	Console I/O routines are available
NORDONLY[1]	EXE$INIT	Executive images should not be set read-only
PFN_INIT	EXE$INIT	Page frame number (PFN) database is initialized
SPNLCK_AVAIL	SYNCH$INIT_ONCE	Spinlock database is available
SWAPPER	EXE$SWAPINIT	Swapper process has begun, and paging is possible
SYSINIT	SYSINIT	SYSINIT process has begun
RMS	SYSINIT	RMS has been loaded
XQP	SYSINIT	File system has been loaded
STARTUP	SYSINIT	Startup process has been created

[1] This bit represents a condition rather than a substage of booting.

32.6.1 Actions of LDR$LOAD_IMAGE

LDR$LOAD_IMAGE must effectively activate an executive image and establish connections between the symbol vector entries and pointers in SYS.EXE and their targets in the loaded image. This section describes the basic operations of LDR$LOAD_IMAGE, with some details of the differences that arise from its execution in different initialization stages.

LDR$LOAD_IMAGE is called with the name of an executive image and a set of flags to control its actions. The symbolic values of these flags, which are defined by the macro $LDRDEF, and their meanings are as follows:

- LDR$V_PAG—When set, indicates that the image should be loaded with its pageable sections resident. The flag is generally based on the value of bit S0PAGING$V_EXEC (bit 0) of the SYSGEN parameter S0_PAGING.
- LDR$V_UNL—When set, indicates that the image may be removed from memory.
- LDR$V_OVR—When set, indicates that the image's read-only sections should not be overwritten during bugcheck processing. This flag is currently not used but is provided for compatibility with VAX/VMS load requests.
- LDR$V_USER_BUF—When set, indicates that the caller has passed the address of a buffer that should be passed to the image's initialization procedure.
- LDR$V_NO_SLICE—When set, indicates the image's sections should not be loaded into a huge page.

LDR$LOAD_IMAGE takes the following steps:

1. It opens the image file using whatever mechanism is available at this stage, either minimal file system routines or the full file system. A WCB is created for a file opened with the minimal file system routines. Later, after SYSINIT has loaded the file system and RMS, SYSINIT opens the file and leaves it open so that, for example, normal file system checks will prevent the file's deletion.

   Running in process context and after system initialization is complete, LDR$LOAD_IMAGE in module SYSLDR_DYN is entered in executive mode and uses RMS to open the image. It then requests the Change to Kernel Mode ($CMKRNL) system service and performs the rest of its processing in kernel mode.

2. LDR$LOAD_IMAGE reads the image header of the file.

3. LDR$LOAD_IMAGE allocates an LDRIMG and initializes it, copying information from its arguments and the image header, such as the procedure value of the initialization procedure and link time. Depending on the boot stage, it performs one of the following:

   — If LDR$LOAD_IMAGE is running as part of SYSBOOT.EXE before nonpaged pool allocation is possible, it builds the LDRIMG in local storage and places it on a local list. Called after nonpaged pool initialization is complete, LDR$LOAD_IMAGE copies each local LDRIMG to nonpaged pool and inserts the nonpaged pool copy at the front of the systemwide LDRIMG list. For each such local LDRIMG, it increments EXE$GL_LDR_

CNT, copies the contents of EXE$GL_LDR_SEQ to LDRIMG$L_SEQ as that image's sequence number, and adds the value 2 to EXE$GL_LDR_SEQ.

— If LDR$LOAD_IMAGE is running after pool allocation is possible but not as part of SYSLDR_DYN.EXE, it allocates an LDRIMG from nonpaged pool and inserts it at the front of the systemwide LDRIMG list. It increments EXE$GL_LDR_CNT, copies the contents of EXE$GL_LDR_SEQ to LDRIMG$L_SEQ as the image's sequence number, and adds the value 2 to EXE$GL_LDR_SEQ.

— If LDR$LOAD_IMAGE is running as part of SYSLDR_DYN.EXE, it allocates an LDRIMG from nonpaged pool and locks the base image mutex, EXE$GQ_BASIMGMTX, for write access. It searches the LDRIMG list to see if an executive image with the same name has already been loaded or is being loaded. If one exists and its flags indicate that it is valid, being unloaded, or being loaded by another process, LDR$LOAD_IMAGE deallocates the LDRIMG, unlocks the mutex, and returns the error status SS$_DUPLNAM to its caller. If one does not, LDR$LOAD_IMAGE sets bit LDRIMG$V_SYNC to indicate that image loading is not complete, inserts the LDRIMG at the front of the LDRIMG list, and unlocks the mutex.

4. It tests BOOSTATE$V_EXEC_SLICING and its argument flags to determine whether the executive image should be loaded sliced and, if not, sets LDRIMG$V_NO_SLICE in LDRIMG$L_FLAGS. By default most executive images loaded prior to the end of system initialization are loaded sliced. It initializes various other bits in LDRIMG$L_FLAGS based on argument flags.

5. LDR$LOAD_IMAGE tests LDRIMG$V_NO_SLICE to see whether the image is to be loaded sliced. If so, it proceeds with step 6.

   Otherwise, scanning the image section descriptors in the image header, LDR$LOAD_IMAGE initializes the appropriate LDRIMG fields to describe each section: its size in bytes, starting VBN within the image file, and starting and ending virtual offsets of the section within the image file. For example, it initializes the fields LDRIMG$L_NONPAG_W_LEN, LDRIMG$L_NONPAG_W_VBN, LDRIMG$L_NONPAG_W_OFFSET, and LDRIMG$L_NONPAG_W_END to describe the resident writable section.

   It calculates the number of pages of address space required to map the image and stores the number in LDRIMG$L_PAGE_COUNT.

   LDR$LOAD_IMAGE allocates contiguous SPTEs for the pages of all the image sections (see Section 32.9.1). It computes the system address represented by the lowest SPTE as the base address of the image, stores it in LDRIMG$L_BASE, relocates the virtual offset of each section by the base address of the image, and updates the corresponding LDRIMG fields. It proceeds with step 7.

6. For an image that is to be loaded sliced, LDR$LOAD_IMAGE scans the image section descriptors in the image header to initialize LDRIMG fields. It allocates address space for the sections as follows:

   — For the nonpaged read-only and writable sections, it allocates slices from either the code or data huge page. If the allocation fails for lack of room, it disables further sliced loading by clearing BOOSTATE$V_EXEC_SLICING, returns whatever resources were allocated, and processes the image as described in step 5.

   — If the flag LDRIMG$V_NPAGED_LOAD is clear (its default state), LDR$LOAD_IMAGE allocates SPTEs for each pageable section. If the flag is set (typically because SYSGEN parameter S0_PAGING is nonzero), it allocates slices from the code or data huge page.

   — For both the initialization and fixup sections, it allocates SPTEs.

   LDR$LOAD_IMAGE records the base virtual address of each image section in the LDRIMG.

7. If the image is either of the two base images, LDR$LOAD_IMAGE stores the relocated start and end virtual addresses of its symbol vector in LDRIMG$L_SYMVVA and LDRIMG$L_SYMVVA_END.

8. It relocates the address of the initialization routine.

9. Depending on whether the image is being loaded sliced, LDR$LOAD_IMAGE invokes either LDR$LOAD_NONPAGED or LDR$LOAD_SLICE, in module SYSLDR_COMMON. It invokes the routine twice—once to map and load the nonpaged read-only code section and once to map and load the writable section. Section 32.6.2 describes LDR$LOAD_NONPAGED, and Section 32.6.3, LDR$LOAD_SLICE.

10. It tests whether the paged image sections are to be loaded as nonpaged. If the LDRIMG$V_NPAGED_LOAD flag is clear, LDR$LOAD_IMAGE invokes LOAD_PAGED, in module SYSLDR_COMMON, once to map the pageable read-only section and once to map the pageable writable section. If the flag is set, LDR$LOAD_IMAGE instead invokes either LDR$LOAD_NONPAGED or LDR$LOAD_SLICE. Section 32.6.4 describes LOAD_PAGED.

11. LDR$LOAD_IMAGE invokes LDR$LOAD_NONPAGED to map and load the fixup section and again to map and load the initialization section. (Even if the image is being loaded sliced, these transient sections are not loaded into a huge page.)

12. LDR$LOAD_IMAGE performs relocations and fixups on references within the loaded image:

    a. It calls LDR$RELOCATE_IMAGE, in module LDR_RELOCATE_FIXUP, to relocate the image's absolute references to its own locations so that they reflect the addresses of the locations in the loaded image. LDR$RELOCATE_IMAGE relocates references within nonpageable sections and, if paging is possible, within the pageable writable section as well. It sets bit LDRIMG$V_NONPAGED_RELOC in LDRIMG$L_

FLAGS. If paging is possible and it can relocate references to pageable sections, or if there are no such references, it sets bit LDRIMG$V_PAGED_RELOC as well.

b. In the probable event that the image itself references SYS.EXE symbols or contains resolutions for any SYS.EXE vectored universal symbols, the image's fixup section contains a shareable image list entry for SYS.EXE. If the image requests system services, its fixup section also contains a shareable image list entry for SYS$PUBLIC_VECTORS.EXE. These entries are built by the linker and must be updated to reflect addresses at which the base images were loaded. LDR$LOAD_IMAGE updates the address of the loaded symbol vector corresponding to each shareable image list entry.

c. LDR$LOAD_IMAGE calls LDR$FIXUP_IMAGE, in module LDR_RELOCATE_FIXUP, to perform fixups, that is, to store correct values in the image's locations that reference external addresses, namely references to locations in SYS.EXE and SYS$PUBLIC_VECTORS.EXE. LDR$FIXUP_IMAGE fixes up references within nonpageable sections and, if paging is possible, within the pageable writable section as well. It sets bit LDRIMG$V_NONPAGED_FIXUPS in LDRIMG$L_FLAGS to indicate that those fixups have been done. If paging is possible, it sets bit LDRIMG$V_PAGED_FIXUPS as well.

   When fixing up references for an image that cannot be unloaded, LDR$LOAD_IMAGE checks whether a second pass fixup is necessary for each SYS.EXE reference being fixed up. If the reference is to a procedure that is still bound to EXE$LOAD_ERROR, LDR$FIXUP_IMAGE records information about the symbol and reference in a nonpaged pool structure pointed to by LDRIMG$L_SPFVA. Section 32.7.2 describes second pass fixups.

d. LDR$LOAD_IMAGE then scans the executive image's global symbol table, looking for vectored universal symbols. As previously described, a vectored universal symbol has two values: the offset of its entry in SYS.EXE's symbol vector and its relative offset within the executive image. For each vectored universal symbol, LDR$LOAD_IMAGE calculates both its address in the loaded SYS.EXE symbol vector and its address based on its offset within the loaded image.

   For a vectored universal data cell, LDR$LOAD_IMAGE updates the data cell pointer in SYS.EXE to point into the loaded executive image.

   For a vectored universal procedure, LDR$LOAD_IMAGE must update its entry in the SYS.EXE symbol vector and its SYS.EXE bound procedure descriptor. LDR$LOAD_IMAGE first determines whether it can access the executive image's procedure descriptor corresponding to the universal symbol. If the procedure descriptor is in a pageable section but paging is not yet possible, LDR$LOAD_IMAGE defers the update. Instead, it stores the information in a piece of nonpaged pool pointed to by LDRIMG$L_GSTVA.

If the procedure descriptor can be accessed and if the executive image, once loaded, may not be removed, then for each vectored universal procedure or routine, LDR$LOAD_IMAGE updates SYS.EXE's symbol vector and binds the SYS.EXE procedure descriptor to the one in the executive image. If the image may be removed, LDR$LOAD_IMAGE merely binds the SYS.EXE procedure descriptor to the one in the executive image.

e. LDR$LOAD_IMAGE executes an instruction memory barrier instruction (see Chapter *Synchronization Techniques*) to flush any prefetched instructions and to ensure that the symbol vector is rewritten before any subsequent writes to memory.

13. LDR$LOAD_IMAGE tests whether the fixup data can be deallocated, that is, whether all needed paged relocations and fixups have been done. If not, it proceeds with the next step. Otherwise, it also tests whether the PFN database has been initialized and, if not, proceeds with the next step. If the PFN database has been initialized, the routine processes the SPTEs that map the fixup section. It deallocates the physical page, if any, associated with each SPTE; it updates the PFN database accordingly; it increments PFN$GL_PHYPGCNT to indicate one more available page of memory; and it clears the SPTE.

    LDR$LOAD_IMAGE triggers invalidation of its processor's TB entries.

    It clears the fields in the LDRIMG that describe the fixup section, subtracts its pages from LDRIMG$L_PAGE_COUNT, and invokes LDR$DEALLOC_PT (see Section 32.9.2) to deallocate the SPTEs.

14. If LDR$LOAD_IMAGE is running as part of SYSBOOT.EXE and there is an initialization procedure, it sets the flag LDRIMG$V_DELAY_INIT in LDRIMG$L_FLAGS so that the procedure will be called at a later stage of initialization. If it is not running as part of SYSBOOT.EXE and the image has an initialization procedure, LDR$LOAD_IMAGE invokes LDR$INIT_SINGLE to call the procedure (see Section 32.7.1).

15. If it is not running as part of SYSLDR_DYN.EXE, it sets LDRIMG$V_VALID to indicate that loading is complete.

16. If LDR$LOAD_IMAGE is running after system initialization is complete, as part of SYSLDR_DYN.EXE, it locks the base image mutex and clears LDRIMG$V_SYNC and sets LDRIMG$V_VALID to indicate that loading is complete. It also increments EXE$GL_LDR_CNT, copies the contents of EXE$GL_LDR_SEQ to LDRIMG$L_SEQ as the image's sequence number, and adds the value 2 to EXE$GL_LDR_SEQ. It unlocks the base image mutex and returns from the kernel mode procedure.

17. LDR$LOAD_IMAGE returns to its caller.

32.6.2 Actions of LDR$LOAD_NONPAGED

LDR$LOAD_IMAGE invokes LDR$LOAD_NONPAGED to map and load non-pageable executive image sections. Its arguments specify the address of the LDRIMG; base, length, and virtual block number of the section; and protection for the section's pages.

LDR$LOAD_NONPAGED performs the following steps:

1. If the spinlock database has been initialized, LDR$LOAD_NONPAGED acquires the MMG spinlock.
2. It calculates the number of pages in the section and the address of the SPTE that maps the first page of the section.
3. For each page of the section, it does the following:
   a. It allocates a page of physical memory.
   b. It initializes the SPTE for that section page with the allocated PFN, owner mode of kernel, valid and address space match bits set, and a protection of URKW. The page must be writable so that it can be overwritten with the contents of the image file. Its protection is changed later.
   c. If the PFN database has been initialized, LDR$LOAD_NONPAGED updates its information about the page, recording information such as the address of the SPTE that contains it, and its state and type. It decrements PFN$GL_PHYPGCNT, the number of physical pages available.

      If the PFN database has not been initialized, recording this information is deferred until EXE$INIT executes.
4. If it has acquired the MMG spinlock, it releases the spinlock.
5. It reads the image section into the allocated space. If there are any errors reading the section, it returns to its invoker.
6. It changes the protection in the section's SPTEs to the protection specified by LDR$LOAD_IMAGE. Its value depends on what boot stage this is and whether the section is writable:
   — If LDR$LOAD_IMAGE is running at a stage before the swapper process, read-only sections are left writable and their pages have a protection of URKW. Otherwise, their protection is UR.
   — Writable section pages have a protection of URKW.

   If LDR$LOAD_NONPAGED is executing after the spinlock database has been initialized, this operation requires that the MMG spinlock be held. LDR$LOAD_NONPAGED triggers invalidation of its processor's TB entries.
7. It returns to its invoker.

32.6.3 Actions of LDR$LOAD_SLICE

LDR$LOAD_SLICE is invoked with the same arguments as LDR$LOAD_NONPAGED. One key difference between the two routines is that memory has already been allocated in the case of a sliced section. Another key difference is that if the granularity hint region has already been set read-only, the pages that make up the slice must first be double-mapped temporarily. The pages are double-mapped by SPTEs that permit write accesses to enable the section to be loaded into memory.

LDR$LOAD_SLICE performs the following steps:

1. It tests whether its protection argument is URKW. That value implies either that the section is to be loaded into a data huge page, which does permit write access, or that LDR$LOAD_SLICE is running at a boot stage where the code page is still writable. (At a later stage, its protection is changed to prohibit writes.) In either case, no double mapping is required, and LDR$LOAD_SLICE continues with step 3.
2. If double mapping is required, LDR$LOAD_SLICE acquires the MMG spinlock and gets the starting physical address of the slice allocated to this section. It allocates enough available SPTEs to map the section and initializes them with appropriate PFNs and a protection of KRKW. It saves the contents of LDRIMG$L_ISD_BASE and replaces them with the system virtual address corresponding to the temporary mapping. It releases the MMG spinlock.
3. It reads the section into the address space specified by LDRIMG$L_ISD_BASE. LDRIMG$L_ISD_LEN contains the length in bytes of the section. The pages that map the section have been initialized with a protection of KRKW so that they can be overwritten with the contents of the image file.
4. If the section was doubly mapped, it deletes the double mapping, restores the former contents of LDRIMG$L_ISD_BASE, and releases the spinlock.
5. It returns to its invoker.

32.6.4 Actions of LOAD_PAGED

LOAD_PAGED is invoked with the same arguments as LDR$LOAD_NONPAGED. It performs the following steps:

1. LOAD_PAGED checks that the BOOSTATE$V_INIT stage of bootstrap has been reached and returns immediately if not. As a result, no image containing pageable sections can be loaded by SYSBOOT.
2. Next it forms prototype PTE contents suitable for mapping each page of the section. The protection, passed as an argument, is either UR for a read-only or URKW for a writable section. The page owner is kernel mode. The type bits in the PTE are set to indicate that the page is part of a section and currently in the image file. If the section is writable, its PTEs also have the copy-on-reference bit set to ensure that page file backing store is allocated for the modified pages.

3. LOAD_PAGED tests and sets the shared bit in the image's WCB. If the bit was clear (if the file had been opened with primitive file routines), LOAD_PAGED initializes its reference count to 2. These steps make the WCB look like any other WCB describing a section file, even if it had been created by primitive file routines, and ensure that the file is permanently open.
4. LOAD_PAGED, running as part of SYSLDR_DYN.EXE, locks the global section mutex for write access.
5. It allocates and initializes a section table entry from the system header (see Chapter *Memory Management Data Structures*).
6. LOAD_PAGED, running as part of SYSLDR_DYN, unlocks the global section mutex.
7. LOAD_PAGED stores the index number of the section table entry in the prototype PTE contents. It records information such as the WCB address, number of pagelets, section base system virtual page number, and a flag indicating whether the section is writable.
8. It records the number of section pages in SEC$L_REFCNT and charges them against the system header's PHD$L_PPGFLVA.
9. LOAD_PAGED writes the prototype PTE to each of the SPTEs previously allocated for the section by LDR$LOAD_IMAGE. If the section does not contain an integral number of *censored* pages, LOAD_PAGED sets the partial section bit in the PTE that maps the last page. The section's pages will be read in later from the executive image in response to page faults, possibly during image initialization when address fixups are done or later during image execution.
10. It returns to LDR$LOAD_IMAGE.

32.6.5 Loading of Optional Images

If the value of bit 0 (SGN$V_LOAD_SYS_IMAGES) of the SYSGEN parameter LOAD_SYS_IMAGES is 1, its default, the loading of optional images is enabled. The images to be loaded are listed in the file SYS$LOADABLE_IMAGES:VMS$SYSTEM_IMAGES.DATA. Each entry specifies the name of an executive image and in which phase, EXE$INIT or SYSINIT, the image should be loaded.

This mechanism provides for the loading of

- Optional VMS-supplied executive images
- Executive images that are part of optional software products
- Site-specific images containing custom versions of the Magnetic Tape Accessibility ($MTACCESS) and Get Security Erase Pattern ($ERAPAT) system services

Both EXE$INIT and SYSINIT call LDR$ALTERNATE_LOAD, in module ALTERNATE_LOAD. LDR$ALTERNATE_LOAD takes the following steps:

1. It tests bit 0 of LOAD_SYS_IMAGES. If the value is zero, the procedure returns.

2. Otherwise, it opens and reads SYS$LOADABLE_IMAGES:VMS$SYSTEM_IMAGES.DATA.
3. For each record in the file, LDR$ALTERNATE_LOAD tests whether it is running during the specified initialization phase. If it is not, LDR$ALTERNATE_LOAD reads the next record.
4. If the current initialization phase matches that in the record, LDR$ALTERNATE_LOAD opens the specified image and reads its image header. It then calls LDR$LOAD_IMAGE to map the image.
5. When LDR$ALTERNATE_LOAD reaches the end of the file, it closes the file and returns to its caller.

The manual *Introduction to VMS System Services* documents the procedure for building a site-specific version of the $MTACCESS system service. Apart from replacements for the $ERAPAT and $MTACCESS system services, use of this mechanism is reserved to Digital.

32.7 Initialization of an Executive Image

Executive image initialization procedures perform a variety of functions, some specific to the features and functions supported by the image, and others required by many executive images.

An initialization procedure may need to execute in an environment that does not exist when the procedure's executive image is first loaded. The initialization mechanism, therefore, provides for delayed and multiple invocations of initialization procedures. An initialization procedure can be called again, for example, so that it can execute after the PFN database has been created or once paging is possible. The space occupied by these procedures is deallocated when initialization is complete.

Initialization procedures are described by an initialization table within each executive image. Each table entry is a quadword. The first longword contains the procedure value of an initialization procedure. The second longword contains flags that describe the initialization procedure and its state.

Each executive image that requires execution of initialization procedures is linked with either the module DOINIT or the module DOINIT_UNL. DOINIT_UNL has all the functionality of DOINIT but also provides for procedures to be called if the image is removed from memory.

Each module defines a number of PSECTs, all of which are clustered into the initialization image section. Three of the PSECTs build the initialization table: EXEC$INIT_000 defines its start and names it INI$A_VECTOR_TABLE; EXEC$INIT_001 defines its body; and EXEC$INIT_002 defines its end with an entry of zero.

Modules in an executive image, including DOINIT or DOINIT_UNL itself, make entries in the body of the initialization table by invoking the macro INITIALIZATION_ROUTINE. An entry for a procedure that should be called before others is made with the macro argument PRIORITY specified as 1. As a result, the entry goes into PSECT EXEC$INIT_000. Otherwise, by default, the entry goes

into PSECT EXEC$INIT_001. The other PSECTs and their uses are described in Section 32.7.5.

DOINIT and DOINIT_UNL also include the initialization procedure dispatcher, INI$DOINIT. Each image linked with DOINIT or DOINIT_UNL specifies INI$DOINIT as its transfer address. LDR$LOAD_IMAGE copies the transfer address from the image header and stores its relocated value in the field LDRIMG$L_INIT_RTN.

INI$DOINIT is called multiple times during system initialization. It scans the initialization table and calls the specified procedures. Each procedure can examine the flags in EXE$GL_STATE to identify the current phase of system initialization and determine whether its execution is appropriate.

DOINIT and DOINIT_UNL each invoke INITIALIZATION_ROUTINE to create a table entry for one common initialization procedure used by most executive images: INI$SYSTEM_SERVICE, which performs initialization for any system services in the image. This initialization procedure is described in Section 32.7.5.

The macro $INIRTNDEF defines symbolic values for the flags in the initialization table. The flag INIRTN$V_CALLED, when set, means that INI$DOINIT has called the initialization routine. The flag INIRTN$V_NO_RECALL, when set, means that the initialization routine should not be called again. The use of these flags is described in Section 32.7.4.

32.7.1 Initialization Sequence

LDR$INIT_SINGLE and LDR$INIT_ALL are the routines that trigger executive image initialization. LDR$INIT_SINGLE initializes a single executive image. LDR$INIT_ALL scans the LDRIMG list, which contains image data blocks for the images loaded thus far, and invokes LDR$INIT_SINGLE for each of them. Both these routines are in module SYSLDR_COMMON and linked with SYSBOOT.EXE, EXEC_INIT.EXE, and SYSLDR_DYN.EXE.

These routines are entered multiple times during system initialization:

1. In the case of each image loaded by SYSBOOT, LDR$LOAD_IMAGE sets the flag LDRIMG$V_DELAY_INIT so that the executive image will be initialized at a later stage.
2. Before switching to the standard system control block (SCB), EXE$INIT calls LDR$INIT_ALL to perform initialization of images loaded by SYSBOOT, in particular, the image SYSTEM_DEBUG.EXE, if it is resident.
3. After the PFN database is initialized, EXE$INIT sets a flag in EXE$GL_STATE to indicate its initialization and calls LDR$INIT_ALL again to perform further initialization of the images loaded by SYSBOOT.
4. EXE$INIT loads the set of executive images listed in Chapter *Bootstrap Processing* by repeatedly calling LDR$LOAD_IMAGE, which invokes LDR$INIT_SINGLE.
5. EXE$INIT then calls LDR$INIT_ALL to perform further initialization of all the images loaded thus far. This additional initialization is done in case actions in one image's initialization routine depend on actions taken in another image's initialization routine.

6. The swapper process sets a flag in EXE$GL_STATE to indicate that it is running and that paging is possible. It initializes paged pool and calls LDR$INIT_ALL. Now that paging is possible, address relocations and external fixups in pageable sections of executive images can be done and system services can be connected.
7. The SYSINIT process loads each of several executive images by calling LDR$LOAD_IMAGE, which invokes LDR$INIT_SINGLE.

In addition, LDR$INIT_SINGLE is invoked to initialize an executive image loaded dynamically after system initialization is complete.

32.7.2 Actions of LDR$INIT_ALL

LDR$INIT_ALL scans the LDRIMG list, from the end of the list to the beginning. This ensures that initialization procedures are called in the order in which their images were loaded.

For each LDRIMG, it performs any relocations and fixups that can be done and then calls the initialization procedure through the following steps:

1. It tests whether the image has a fixup section and, if not, proceeds to step 4. If there is one, it represents relocations and fixups in pageable sections. LDR$INIT_ALL tests whether paging is possible and, if not, defers the relocations and fixups and proceeds to step 3.
2. If the image has a fixup section and paging is possible, LDR$INIT_ALL takes the following steps to perform relocation and fixup:
   a. If absolute references within the pageable writable section of the image have not been relocated, LDR$INIT_ALL calls LDR$RELOCATE_IMAGE, in module LDR_RELOCATE_FIXUP, to relocate them so that they reflect the addresses of the locations in the loaded image.
   b. If pageable fixups remain to be done, LDR$INIT_ALL calls LDR$FIXUP_IMAGE, in module LDR_RELOCATE_FIXUP, to perform external fixups.
   c. If there are still SYS.EXE symbol vector entries to be replaced, LDR$INIT_ALL processes each entry in the nonpaged pool structure whose address is in LDRIMG$L_GSTVA.

      Each entry represents a vectored universal symbol that is the name of a procedure or routine whose procedure descriptor is pageable and thus inaccessible during early stages of system initialization. Each entry contains the address of its SYS.EXE symbol vector entry and the address of its executive image procedure descriptor.

      LDR$INIT_ALL examines the contents of the SYS.EXE symbol vector entry to get the address of the bound procedure descriptor for that vectored universal symbol and binds it to the procedure descriptor in the executive image. Unless the executive image is removable, LDR$INIT_ALL replaces the SYS.EXE symbol vector entry contents with the addresses of the procedure and its code entry point. When processing is done, LDR$INIT_ALL deallocates the nonpaged pool structure, clears LDRIMG$L_GSTVA, and clears LDRIMG$V_GSTVA_VALID.

3. LDR$INIT_ALL tests whether the fixup data can be deallocated, that is, whether all needed paged relocations and fixups have been done. If not, it proceeds with the next step. Otherwise, it also tests whether the PFN database has been initialized and, if not, proceeds with the next step.

   If the PFN database has been initialized, the routine processes the SPTEs that map the fixup section. For each such page, it acquires the MMG spinlock. It determines the PFN occupied by the virtual page and changes its PFN database to reflect its new unused status. It increments PFN$GL_PHYPGCNT to show one more page of available memory, invokes MMG$DALLOC_PFN to release the page, clears the SPTE, and releases the MMG spinlock.

   If the executive image containing the invalidate code has been loaded (if BOOSTATE$V_SWAPPER in EXE$GL_STATE is set), LDR$INIT_ALL triggers invalidation of its processor's TB entries.

   It clears the fields in the LDRIMG that describe the fixup section, subtracts its pages from LDRIMG$L_PAGE_COUNT, and invokes LDR$DEALLOC_PT (see Section 32.9.2) to deallocate the SPTEs.

4. LDR$INIT_ALL tests whether the initialization procedure should be called again. If so, LDR$INIT_ALL invokes LDR$INIT_SINGLE (see Section 32.7.3) to call the initialization procedure.

After processing the entire list, LDR$INIT_ALL performs second pass fixups. It scans the LDRIMG list. For each image, it takes the following steps:

1. It tests whether any second pass fixups need to be performed for this image. If not, it proceeds to the next image.

   If second pass fixups need to be done, LDRIMG$L_SPFVA points to a nonpaged pool structure that contains a list of fixup entries. Each entry in the structure represents a linkage pair, procedure descriptor address, or code entry address. Each entry contains an identifying type, the address in the executive image to be fixed up, and the address of the corresponding SYS.EXE symbol vector entry.

2. For each entry in the nonpaged pool structure, LDR$INIT_ALL examines the corresponding symbol vector entry to determine whether it still contains the address of a bound procedure descriptor in SYS.EXE. If not, LDR$INIT_ALL performs the fixup, replacing the reference in the executive image with the appropriate address or addresses in the other executive image.

3. For each fixup performed, LDR$INIT_ALL decrements the number of entries left in the nonpaged pool structure. If there are none left, it deallocates the structure and clears LDRIMG$L_SPFVA and LDRIMG$V_SECOND_PASS_FIXUP.

LDR$INIT_ALL returns to its caller.

32.7.3 Actions of LDR$INIT_SINGLE

LDR$INIT_SINGLE performs the following steps:

1. It tests whether the value of LDRIMG$L_INIT_RTN is zero. If so, the routine proceeds to step 4.
2. Otherwise, it calls the routine whose procedure value is in LDRIMG$L_INIT_RTN. It passes one argument, the address of the LDRIMG. Although this mechanism allows for other possibilities, LDRIMG$L_INIT_RTN currently always contains the procedure value of INI$DOINIT. If the caller of LDR$LOAD_IMAGE specified the address of a user-defined buffer, its address has been passed to LDR$INIT_SINGLE, which passes it to the initialization procedure.
3. If INI$DOINIT returns an error status, LDR$INIT_SINGLE returns to its invoker.
4. LDR$INIT_SINGLE tests the flag LDRIMG$V_DELAY_INIT, which is set by INI$DOINIT when an initialization routine specifies that it must be reinvoked. If the flag is clear, LDR$INIT_SINGLE tests whether the PFN database has been initialized and, if not, returns to its invoker.
5. If the PFN database has been initialized, LDR$INIT_SINGLE tests whether the virtual address space occupied by the initialization section has already been deallocated. If so, it returns to its invoker.
6. Otherwise, it calculates the number of pages of initialization routine and the first SPTE that maps these pages from the fields in the LDRIMG that describe the initialization routine section.
7. For each such page, LDR$INIT_SINGLE acquires the MMG spinlock. It determines the PFN occupied by the virtual page and changes its PFN database to reflect its new unused status. It increments PFN$GL_PHYPGCNT to show one more page of available memory, invokes MMG$DALLOC_PFN to release the page, clears the SPTE, and releases the MMG spinlock.
8. LDR$INIT_SINGLE triggers invalidation of its processor's TB entries.
9. It clears the fields in the LDRIMG that describe the deallocated initialization section. It tests whether the section is at the end of the loaded image and, if so, updates LDRIMG$L_PAGE_COUNT to reflect the deallocation of that address space and invokes LDR$DEALLOC_PT to insert the SPTEs that map the deallocated section onto the free SPTE list. Section 32.9.2 describes the deallocation of system space.
10. It returns to its invoker.

32.7.4 Actions of INI$DOINIT

INI$DOINIT is the initialization procedure dispatcher. It performs the following steps:

1. It clears the LDRIMG flag LDRIMG$V_DELAY_INIT to implement its default of not scanning the initialization table again.
2. It scans the table.

3. For each entry, it tests and sets the INIRTN$V_NO_RECALL flag to implement its default of calling an initialization procedure only once. If the flag was already set, it goes on to the next entry.
4. If the flag was clear, INI$DOINIT calls the initialization procedure.

   If the initialization procedure determines that it should be reentered at a later stage of system initialization, it clears the INIRTN$V_NO_RECALL flag.
5. When the routine returns, INI$DOINIT sets the flag INIRTN$V_CALLED to record that the procedure was called and tests INIRTN$V_NO_RECALL. If the flag is clear, INI$DOINIT sets LDRIMG$V_DELAY_INIT to ensure that LDR$INIT_SINGLE does not deallocate the initialization section and that INI$DOINIT will be recalled at a later stage of system initialization.

32.7.5 Initialization Procedures

An image-specific initialization procedure might do a number of things, including, but not limited to, the following:

- Initialize an SCB entry to describe an interrupt or exception service routine
- Initialize SYS.EXE globals
- Initialize data in the executive image
- Allocate pool for a data structure

Chapter *Operating System Initialization and Shutdown* summarizes the actions of the initialization procedures for various executive images.

One common initialization procedure is part of many executive images: INI$SYSTEM_SERVICE.

INI$SYSTEM_SERVICE connects any system services in the executive image to the system service transfer routines and procedure descriptors in the loaded SYS$PUBLIC_VECTORS.EXE and assigns change mode numbers for inner access mode services. Chapter *System Service Dispatching* describes in detail the relations among SYS$PUBLIC_VECTORS.EXE's symbol vector, system service transfer routines, their procedure descriptors, and loaded services in executive images.

INI$SYSTEM_SERVICE is table-driven, using INI$A_BUILD_TABLE, a table of system service descriptor blocks within the image. The module DOINIT defines three PSECTs that build the table: EXEC$INIT_SSTBL_000 defines its start; EXEC$INIT_SSTBL_001 defines its body; and EXEC$INIT_SSTBL_002 defines its end with an entry of zero. Modules in an executive image make entries in the body of its table by invoking the macro SYSTEM_SERVICE. The macro initializes the fields in a system service descriptor block.

Each system service descriptor block contains the following fields:

- The address of the system service transfer routine's procedure descriptor in SYS$PUBLIC_VECTORS.EXE
- The address of the procedure descriptor for the system service-specific procedure within the executive image

- Flags indicating whether the service can return the status SS$_WAIT_CALLERS_MODE and a flag indicating whether the scratch registers should be cleared after the service exits
- The system service filter group
- A code indicating the access mode in which the system service procedure executes

The macro $SSDESCRDEF defines symbolic offsets for these fields, and Chapter *System Service Dispatching* explains the significance of their contents.

INI$SYSTEM_SERVICE takes the following steps:

1. Because INI$SYSTEM_SERVICE requires process context to execute, it first tests a flag in EXE$GL_STATE to determine whether the swapper process has begun to execute. If not, the procedure clears INIRTN$V_NO_RECALL so that the routine will be entered in process context, when paging is possible, and returns.

   If the swapper process has begun to execute, INI$SYSTEM_SERVICE proceeds.

2. It changes the protection on the pages of the loaded SYS$PUBLIC_VECTORS.EXE that contain the system service transfer routines and procedure descriptors, making them writable so that it can modify a service's transfer routine or procedure descriptor.

3. It scans the system service descriptor block table. For each block it finds, it calls EXE$CONNECT_SERVICES, in module SYSTEM_SERVICE_LOADER, passing it the address of the system service descriptor block.

4. When INI$SYSTEM_SERVICE reaches the end of the table, it makes the pages of SYS$PUBLIC_VECTORS.EXE read-only again and returns.

EXE$CONNECT_SERVICES takes the following steps:

1. It first gets the code entry point addresses of the system service transfer routine and the system service-specific procedure from their procedure descriptors.

2. It determines whether the service is kernel mode, executive mode, or mode of caller.

3. For a mode of caller service, it checks that the instructions in the system service transfer routine are correct. If they are not, it generates the fatal bugcheck BADVECTOR.

   If they are correct, EXE$CONNECT_SERVICES searches the loaded symbol vector in SYS$PUBLIC_VECTORS.EXE until it finds the entry whose procedure value matches that of the system service transfer routine. It updates that entry to contain the procedure value and code entry address of the service-specific procedure within the executive image.

   EXE$CONNECT_SERVICES also binds the transfer routine procedure descriptor to that of the service-specific procedure. Subsequently, when an image that requests this service is activated, the linkage pair that represents its service procedure call will be fixed up with information about

the loaded service rather than about the transfer routine. An image activated before service connection executes the transfer routine to dispatch to the loaded service.

4. For an inner access mode service, EXE$CONNECT_SERVICES does the following:
   a. It acquires write ownership of a mutex called the change mode mutex, which prevents multiple processes from adding system services concurrently.
   b. It performs several sanity checks on the system service transfer routine and, if they fail, generates the fatal bugcheck BADVECTOR.
   c. If they are correct, EXE$CONNECT_SERVICES gets the change mode number to be assigned to this system service (from CMOD$GL_CHMK_LIMIT for kernel mode, CMOD$GL_CHME_LIMIT for executive mode) and tests that the number is less than or equal to 255, the maximum number. If it is not, EXE$CONNECT_SERVICES generates the fatal bugcheck SSVECFULL.
   d. If it is less than or equal to 255, EXE$CONNECT_SERVICES constructs the system service transfer routine as an instruction to save the stack pointer (SP), an instruction to load the change mode number into R0, the appropriate change mode instruction, and a RET.
   e. It executes an instruction memory barrier instruction (see Chapter *Synchronization Techniques*) to flush any prefetched instructions in case any of those just rewritten had been prefetched and to ensure that the transfer routine is rewritten before subsequent writes to memory.
   f. EXE$CONNECT_SERVICES records information from the system service descriptor block in arrays used by the change mode dispatchers. One set of arrays describes kernel mode system services; another set describes executive mode services. Each array is indexed by the change mode number of the service. Chapter *System Service Dispatching* describes these arrays and their uses.
   g. EXE$CONNECT_SERVICES increments CMOD$GL_CHM*x*_LIMIT and releases the change mode mutex. (The change mode dispatcher compares the contents of that cell against the number associated with a change mode instruction to test whether it is within an executive image, as described in Chapter *System Service Dispatching*.)
5. It returns to its caller.

32.7.6 Actions of LDR$RELEASE_MEM

LDR$RELEASE_MEM, in module LDR_MEM_ALLOC, is called by image LDR$WRAP.EXE, which runs at the end of system initialization after sliced loading of executive images and installed resident images is complete.

It releases unused memory from huge pages, if appropriate, and completes executive image loading by performing second pass fixups.

It takes the following steps:

1. It locks the base image mutex.
2. It tests bit SGN$V_RELEASE_PFNS in SYSGEN parameter LOAD_SYS_IMAGES to see whether any unused portions of huge pages should be released to the free page list. If the bit is 0, LDR$RELEASE_MEM does not release pages to the free page list and proceeds with step 6.
3. If the bit is 1, its default value, LDR$RELEASE_MEM determines which types of huge page have been created and releases unused portions, if possible.

   — If a huge code page has been created, LDR$RELEASE_MEM optionally reserves an unused portion for later installation of resident images, depending on the SYSGEN parameter SHRIMG_FREEMEM. The parameter specifies the number of kilobytes to be reserved. If the parameter is nonzero, LDR$RELEASE_MEM reserves the larger of the unused portion and the value of the parameter. It deallocates to the free page list any physical pages that were not allocated or reserved and clears their bitmap entries in the LDRHP. It clears the SPTEs that mapped them and triggers invalidation of its processor's TB entries.

     A subsequent attempt to access these unused virtual pages will be translated, even though those pages are invalid, if any TB entry maps a valid page within the huge page. However, the protection on the valid pages in the region will result in faults for any access but attempted execution from such a page through the system virtual address that would have mapped it.

   — LDR$RELEASE_MEM examines the bitmap in the huge data page LDRHP and, if the second 64-page piece is unused, it deallocates those physical pages and SPTEs, as described in the previous step.

     Unused physical pages within a data page granularity hint region, part of which is in use, cannot be deallocated. If they were deallocated and then reallocated for another use, the protection on the SPTEs that map the used pages could enable undesired read or write access to the reallocated pages.

   — It examines the bitmap in the third huge data page and releases any physical pages that are unused. The virtual pages from this granularity hint region that are in use permit read and write access only by inner modes. Thus, attempted access to a reallocated page from an outer mode through the system virtual address that would have mapped it, would cause an access violation.

   Chapter *Memory Management Overview* describes in further detail the implications for address translation of releasing pages from a granularity hint region.
4. LDR$RELEASE_MEM performs any remaining possible second pass fixups and deallocates all nonpaged pool structures pointed to by any executive image's LDRIMG$L_SPFVA field.

5. It releases the base image mutex.
6. It returns to its caller.

32.8 Version Numbers

The intent of dividing the executive into two base images and a number of executive images is to minimize the frequency at which images linked with SYS.EXE must relink. A change to an executive image does not alter the offsets of its vectored universal symbols in SYS.EXE's symbol vector. However, data structure and routine interface changes within an executive image may require algorithmic changes and reassembly of any images using its routines and data.

VMS implements a form of internal system version identifier that can denote data structure and routine interface changes. This number is independent of the external VMS version number. Because the executive organization does not tie a routine to a particular executive image, a version number for each executive image is not a good solution. Instead, the executive is divided into conceptual categories, such as I/O or memory management, each with its own version number. Table 32–3 lists these conceptual categories, each of which is identified by a number. The $SYSVERSIONDEF macro defines symbols for these numbers.

Each SYS.EXE global symbol specifies the conceptual categories with which it is associated, through the VERSION_MASK keyword in the macro that defines the global symbol. Each bit in the mask corresponds to the number of a conceptual category. The macros that define SYS.EXE data cell and routine name symbols can be compiled to generate a mask global for each global.

For example, the routine EXE$ALLOCIRP, invoked to allocate an IRP, is associated with the category MEMORY_MANAGEMENT. The symbol PFN$AL_HEAD is also associated with that category. Extracts from SYSTEM_ROUTINES and SYSTEM_DATA_CELLS that define those symbols and their masks are shown in Examples 32–1 and 32–2.

Each category version number is a longword, with major ID in the high-order word and minor ID in the low-order word. Each is defined by a symbol named SYS$K_*category-name*. The category version numbers are defined in SYS.EXE.

The version number for a category changes when an interface in that category changes. The minor ID changes for an upwardly compatible change; the major ID changes for an incompatible change. For example, if a routine's input arguments or a data structure's fields are redefined, then images referencing that routine or data structure will not execute properly unless they are changed. In such a case, the major ID is incremented. Examples of an upwardly compatible change are the addition of optional arguments to a routine and the use of data structure fields previously defined as spare.

Table 32–3 Executive Version Categories

Category Name	Number	Description
BASE_IMAGE	0	Base image
MEMORY_ MANAGEMENT	1	Memory management and dynamic pools
IO	2	I/O data structures and routines
FILES_VOLUMES	3	RMS and file system
PROCESS_SCHED	4	Process control, scheduling, and structure; layout of P1 space; timer events, ASTs, and event flags
SYSGEN	5	SYSGEN parameters
CLUSTERS_LOCKMGR	6	Reserved
LOGICAL_NAMES	7	Logical names
SECURITY	8	Security subsystem
IMAGE_ACTIVATOR	9	Image activation and image file interpretation
NETWORKS	10	DECnet and support for datalink drivers
COUNTERS	11	Cells that are interpreted as counts
STABLE	12	Routines and data structures expected to be stable
MISC	13	Miscellaneous
CPU	14	CPU-specific support
VOLATILE	15	Routines and data structures expected to change in the next release
SHELL	16	The layout of the SHELL module and P1 space

Four of the modules that make up SYS.EXE are conditionally assembled to generate mask globals:

- SYSTEM_ROUTINES
- SYSTEM_DATA_CELLS
- SYSPARAM
- SHELL

The resulting object modules are concatenated to form a module called BASE_IMAGE.MASK. When SYS.EXE is linked, BASE_IMAGE.MASK is specified in the option statement MASK_TABLE. The linker checks whether each SYS.EXE global symbol is also defined in BASE_IMAGE.MASK. If it is, the linker creates another definition of the global symbol that is its mask value. The mask values are used when some other image links with SYS.EXE.

An image header includes space for an array of category version numbers. The first longword of the array contains a mask identifying which categories are relevant to the image. The image header field EIHD$L_SYSVER contains the overall system version number, with the major version number in the high-order byte and the minor version number in the low-order three bytes. When an image referencing a SYS.EXE masked global symbol is linked, the linker ORs the value

of the corresponding mask global into the image's category mask longword. After all globals have been resolved, the mask has a bit set for each conceptual category relevant to the image. Starting from bit 0, the linker stores the relevant category version number constants from SYS.EXE's symbol vector into the subsequent longwords of the version array. There are no entries in the version array for categories not relevant to that image.

Example 32–4 shows an extract from the output of the DCL command ANALYZE/IMAGE SDA.EXE.

Example 32–4 Extract from Output of ANALYZE/IMAGE SDA.EXE Command

```
.
.
X56E_RESD$:[SDA.OBJ]SDA.EXE;1
ANALYZ T05-01

IMAGE HEADER

 Fixed Header Information

  image format major id: 2, minor id: 0
  header block count: 3
  image type: executable (EIHD$K_EXE)
  I/O channel count: default
  I/O byte count: default
  Symbol Vector Virtual Address: %X'00000000'
  Symbol Vector Size: 0 bytes
  Virtual Memory Block Size: 65536 (BPAGE = 16)
  Fixup Section Virtual Address: %X'00100000'
  linker flags:
   (0)  EIHD$V_LNKDEBUG  0
   (1)  EIHD$V_LNKNOTFR  0
   (2)  EIHD$V_NOP0BUFS  0
   (3)  EIHD$V_PICIMG    1
   (4)  EIHD$V_P0IMAGE   0
   (5)  EIHD$V_DBGDMT    1
   (6)  EIHD$V_INISHR    0
   (7)  EIHD$V_XLATED    0
   (8)  EIHD$V_BIND_CODE 0
   (9)  EIHD$V_BIND_DATA 0
  system version (major/minor): 1.0
  system version array information: (Image Value)
   SYS$K_MEMORY_MANAGEMENT : (1.64)
   SYS$K_PROCESS_SCHED     : (1.64)
   SYS$K_SYSGEN            : (1.64)
   SYS$K_STABLE            : (1.64)
   SYS$K_VOLATILE          : (1.64)
.
.
```

The BASE_IMAGE category describes the layout of SYS.EXE rather than any particular conceptual category. The BASE_IMAGE minor ID is altered when a new universal procedure or data cell is added so that an image using the new symbol cannot run on an older version. An alteration to the BASE_IMAGE major ID forces all images linked with SYS.EXE to be relinked before they can be activated. Required when the layout of SYS.EXE changes, this is expected to be rare.

The overall system version, SYS$K_VERSION, has for its major ID the major ID of the BASE_IMAGE category. Its minor ID represents the particular release or build; its use is reserved to VMS.

SYS.EXE global SYS$GL_VERSION begins a 32-longword array of version numbers generated from the assembly of module VERSION_NUMBERS. When an image linked with SYS.EXE is activated, the routine EXE$CHECK_VERSION, in module CHECK_VERSION, is invoked to compare the array of version numbers in its image header with the versions of the running executive. All of the following must be true:

- The major ID of the image must match the major ID of the running system.
- The minor ID must be less than or equal to that of the running system.
- The first longword of the EIHD version array contains a mask of conceptual executive categories relevant to the image. For each bit set in the mask, the major ID of the executive category at the time the image was linked must be equal to that of the category in the running system. The minor ID must be less than or equal to that of the category in the running system.

If the versions are incompatible, the image activator aborts the activation and returns the fatal error status SS$_SYSVERDIF.

32.9 Dynamic Allocation and Deallocation of SPTEs

VMS implements dynamic allocation and deallocation of SPTEs. This enables more flexible allocation of system virtual address space than would be possible if all system virtual address space were assigned during system initialization. Instead, for example, virtual address space can be allocated for an executive image when it is loaded, and the space occupied by its initialization and fixup sections can be deallocated after they are no longer needed. (Even if the image is loaded sliced, its initialization and fixup sections are not loaded into a huge page.) The freed address space can be reused.

As described in Chapter *Bootstrap Processing* and Appendix *Size of System and P1 Virtual Address Spaces,* SYSBOOT defines the initial size and layout of system space, based largely on SYSGEN parameter values. If there is insufficient system space, it can be expanded toward higher addresses during normal system operation.

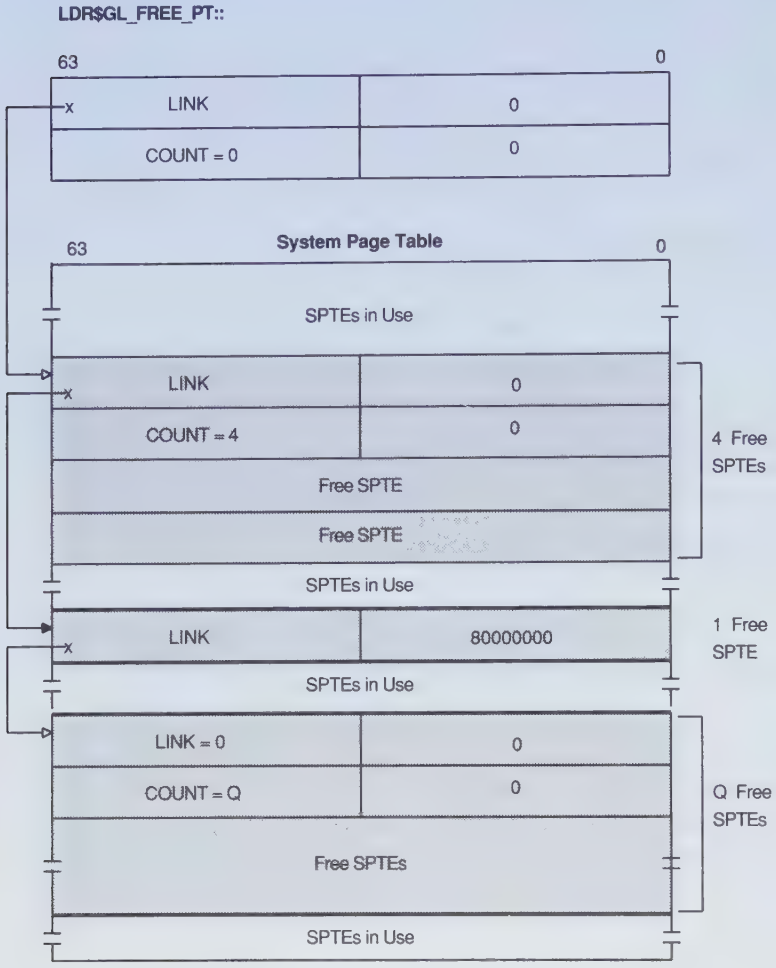
Two routines in module PTALLOC maintain a list of available system pages:

- LDR$ALLOC_PT, which allocates SPTEs
- LDR$DEALLOC_PT, which deallocates SPTEs

Their actions are described in Sections 32.9.1 and 32.9.2.

The list of available pages of system space is kept within the available SPTEs themselves. Its listhead is at global cell LDR$GL_FREE_PT, which points to the first element on the list. Figure 32–8 shows the form of the list, with free SPTEs shaded.

Figure 32–8 List of Available SPTEs



Each element on the list represents a group of adjacent available SPTEs. The smallest group is one SPTE. A single available SPTE contains, in bits <63:32>, a pointer to the next group. Bit 31 is set to identify the SPTE as the sole member of its group.

Two SPTEs are required to describe an element consisting of a group of two or more adjacent available SPTEs. The first SPTE points to the next group of free SPTEs; the second contains the number of SPTEs in this group.

The low-order 16 bits of each free SPTE, which include protection code bits and the valid bit, must be zero so that the SPTE appears to map an invalid page with all access prohibited.

A group of free SPTEs is identified by its quadword index from the beginning of the system page table (SPT), that is, by the virtual page number corresponding to the first free SPTE in that group. The quadword index of the next element is stored in bits <63:32> of the SPTE. (Each SPTE is a quadword.) For example, if LDR$GL_FREE_PT + 4 contains 100_{16}, the first SPTE available for allocation is

at offset (100_{16} * 8) from the base of the SPT. The number of SPTEs in that group is at offset (100_{16} * 8) + C_{16}.

The SPTE allocation algorithm is first-fit and takes the higher end of a group of SPTEs if the group is larger than needed. The SPTE deallocation algorithm orders the list from smaller virtual page number to larger.

Much SPTE allocation occurs during system initialization, in SYSBOOT and EXE$INIT. These execute at interrupt priority level (IPL) 31. When LDR$ALLOC_PT and LDR$DEALLOC_PT are invoked at later stages of initialization, they synchronize their accesses to the SPTE list by acquiring the MMG spinlock.

32.9.1 Actions of LDR$ALLOC_PT

LDR$ALLOC_PT is invoked with the number of SPTEs to be allocated. It takes the following steps:

1. If it is running after the swapper process has begun, it acquires the MMG spinlock, raising IPL to IPL$_MMG.
2. It scans the list of available SPTEs, starting with the group whose offset is stored in LDR$GL_FREE_PT + 4, looking for a large enough group.
3. If LDR$ALLOC_PT finds a group exactly the right size, it removes that group from the list by changing the forward pointer of the predecessor group to point to the next group.
4. If it finds a group larger than needed, it subtracts the number of SPTEs needed from the count longword. If the count is reduced to 1, LDR$ALLOC_PT sets bit 31 in the single available SPTE. It allocates the SPTEs at the high end of the group so that it does not have to copy the pointer and count longwords and to alter the longword pointing to the beginning of the group.
5. It zeros the allocated SPTEs and returns to its invoker the address of the lowest SPTE in the allocated group and a status of SS$_NORMAL.
6. If it cannot make the allocation, either because there are no free SPTEs or no group large enough, LDR$ALLOC_PT attempts to expand the SPT. A [*censored*] VMS SPT can be expanded to map higher addresses, as long as there is an unused level 2 PTE (L2PTE) to map a new page of the SPT. LDR$ALLOC_PT first calculates the L2PTE index corresponding to the highest system virtual address currently defined (the contents of PHD$L_FREP0VA, in the system header).
7. It calculates the number of SPTEs to be added, based on the request size and the number of any contiguous free SPTEs at the end of existing system space. If there are not enough L2PTEs to create that much system space, it returns the error status SS$_INSFSPTS to its invoker, releasing the MMG spinlock and lowering IPL if appropriate.

   The minimum number of SPTEs added at once is the number of SPTEs in a page of a level 3 page table (L3PT).

8. For each page of SPT to be added, LDR$ALLOC_PT allocates a physical page of memory. If it is unable to allocate a page of memory, it returns the error status SS$_INSFSPTS to its invoker, releasing the MMG spinlock and lowering IPL if appropriate.

   If allocation was successful, LDR$ALLOC_PT updates the PFN database for that page to reflect its new status as an active system page. LDR$ALLOC_PT decrements PFN$GL_PHYPGCNT, the number of fluid physical pages. It initializes the L2PTE with the PFN of the physical page just allocated; a protection permitting kernel reads and writes; and valid, address space match, and fault-on-execute bits set. Chapter *Memory Management Data Structures* describes the PFN database and the address translation algorithm.

9. It clears the newly allocated SPTEs, forms them into a group, and inserts the group on the free SPTE list. If the new SPTEs are adjacent to the last group on the list, LDR$ALLOC_PT merges them into that group.

10. LDR$ALLOC_PT allocates the requested SPTEs.

11. It updates PHD$L_FREP0VA and MMG$GL_FRESVA to reflect the new high end of system space and PHD$L_P0LENGTH, the new length in bytes of the SPT.

12. It releases the MMG spinlock, if it has been acquired, lowering IPL.

32.9.2 Actions of LDR$DEALLOC_PT

LDR$DEALLOC_PT is invoked with the address of the lowest SPTE in the group to be deallocated and the number of SPTEs in the group. The invoker must have already deallocated any physical memory associated with the SPTEs, invalidated cached TB entries, and zeroed the SPTEs.

LDR$DEALLOC_PT takes the following steps:

1. If it is running after the swapper process has begun, it acquires the MMG spinlock, raising IPL to IPL$_MMG.

2. It first checks that the SPTEs are all zero. If they are not, it releases the spinlock, lowering IPL, and returns the error status LOADER$_PTE_NOT_EMPTY to its invoker.

3. Otherwise, it scans the list of available SPTEs, looking for the first group whose address is higher than that of the group being deallocated.

4. It inserts the group being deallocated at that point and checks whether it can be merged with the group on either side of it. It makes whatever merges are possible, altering pointers and count longwords as appropriate.

5. It releases the spinlock, if it has been acquired, lowering IPL, and returns the status SS$_NORMAL to its invoker.

32.10 Revision Information

Source Module	Build Date
[LIB]BOOSTATEDEF.SDL	20-Feb-1992
[LIB]LDRIMGDEF.SDL	20-Feb-1992
[LIB]LDRHPDEF.SDL	20-Feb-1992
[LIB]VECTORS.SDL	20-Feb-1992
[SYS]DOINIT.MAR	20-Feb-1992
[SYS]PTALLOC.MAR	20-Feb-1992
[SYS]LDR_MEM_ALLOC.B64	20-Feb-1992
[SYS]LDR_MEM_INIT.B64	20-Feb-1992
[SYS]LDR_RELOCATE_FIXUP.B64	20-Feb-1992
[SYS]SYSLDR.MAR	20-Feb-1992
[SYS]SYSLDR_COMMON.MAR	20-Feb-1992
[SYS]SYSLDR_DYN.MAR	20-Feb-1992

Chapter 33

Bootstrap Processing

Ante mare et terras et quod tegit omnia caelum unus erat toto
naturae vultus in orbe, quem dixere chaos: rudis indigestaque moles.
[Before the sea was, and the lands, and the sky that hangs
over all, the face of Nature showed all alike, which state has
been called chaos: a rough unordered mass of things.]

Ovid, *Metamorphoses* I, 5–7

The process by which the VMS operating system assumes control of a [*censored*] system is called system initialization. System initialization occurs in two phases. In the first phase, a series of stand-alone programs (sometimes called bootstrap programs) initialize system hardware blocks and load part of the VMS executive into memory. In the second phase, the rest of the VMS executive is loaded and initialized.

This chapter describes the first phase of system initialization. Chapter *Operating System Initialization and Shutdown* describes the next phase of system initialization.

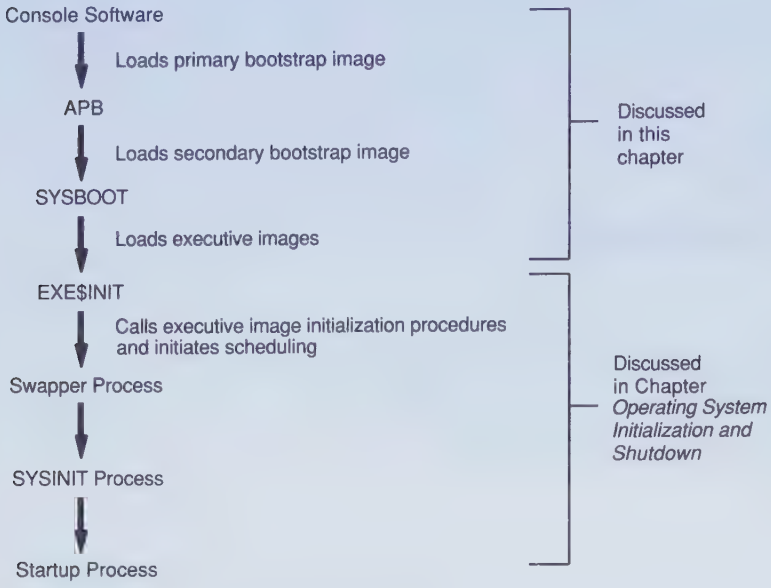
33.1 Overview of System Initialization

VMS system initialization requires a number of programs. Some of them run prior to the establishment of an operating system environment; others execute in system context; others execute in process context. In general, VMS postpones each initialization task to as late a stage of initialization as possible, because initialization is progressively easier as more of the operating system becomes available.

The sequence of operations that occurs during VMS system initialization is shown in Figure 33–1 and outlined in the following list:

1. The console subsystem initializes the processor; locates enough good memory for APB, the [*censored*] primary bootstrap program; sets up APB's initial environment; loads privileged architecture library code (PALcode); loads APB; and transfers control to it. Section 33.2 summarizes the console's role in system initialization.

Figure 33–1 System Initialization Overview



2. APB runs stand-alone at interrupt priority level (IPL) 31. It tests and sizes more memory if necessary, loads the secondary bootstrap program, and transfers control to it. Section 33.3 describes APB.

3. SYSBOOT, the secondary bootstrap program for the VMS system, also runs stand-alone at IPL 31. Based on parameters from the system generation (SYSGEN) utility parameter file and the console operator, it creates the system page table (SPT) and maps system virtual address space. It loads into memory the system base images and several executive images.

   SYSBOOT then transfers control to EXE$INIT, the executive initialization procedure. Section 33.4 describes SYSBOOT in detail.

4. Running at IPL 31, EXE$INIT, in the executive image EXEC_INIT, deletes bootstrap address space. Operating out of system space, it loads most of the remaining executive images and calls the initialization procedures of all loaded executive images. The executive images initialize the scheduler, memory management, spinlock, and I/O databases, among other operations. EXE$INIT then calls SCH$INIT, the scheduler routine that lowers IPL from 31 to IPL$_RESCHED and initiates scheduling. SCH$INIT places the swapper process into execution (see Chapter *Operating System Initialization and Shutdown*).

5. Initialization code that runs in the context of the swapper process performs the minimum tasks that must complete in process context before any other processes can be created. Because it is pageable code, it eventually disappears from the system working set and thus occupies no physical memory. Its tasks include initializing paged pool and the pageable logical name database, and invoking executive image initialization routines that require process context to execute. The swapper creates the SYSINIT process.

6. The SYSINIT process performs initialization tasks that must be done in process context and that do not lend themselves to Digital command language (DCL) commands. These include initializing the swap and page files and creating the startup process.
7. The startup process maps DCL and can thus execute a series of DCL commands. It executes the command procedure SYS$SYSTEM:STARTUP.COM, which processes other command procedures and data files in the SYS$STARTUP directory. The various command procedures create system processes, such as OPCOM and JOB_CONTROL. They create systemwide logical names, use the IOGEN utility to autoconfigure the I/O database, and install images specified by the VMSIMAGES.DAT data file. The startup process executes a series of site-specific command procedures and finally enables interactive login.

From SYSBOOT onward, the files and programs used in bootstrap operations are primarily independent of processor type. Table 33–1 lists bootstrap programs and processes used during system initialization, the files they process, and the function of each file.

Table 33–1 Files and Programs Used During System Initialization

Files Accessed	Function of File
	SYSBOOT
ERRORLOG.EXE	Error logging routines and system services
EXEC_INIT.EXE	Next image in bootstrap sequence
PAGEFILE.SYS	Primary page file, located and sized if dump file not found
PRIMITIVE_IO.EXE	Primitive console I/O and file system routines
SYS.EXE	System base image
SYSDUMP.DMP	System dump file, located and sized for later use
SYSTEM_DEBUG.EXE	System debugger (XDELTA), conditionally loaded
SYSTEM_PRIMITIVES_*xxx*.EXE	Basic system support routines (one of two—see Chapter *Pool Management*)
SYSTEM_ SYNCHRONIZATION_*xxx*.EXE	Symmetric multiprocessing (SMP) synchronization image (one of three—see Chapter *Synchronization Techniques*)
SYS$CPU_ROUTINES_ *xxyy*.EXE	CPU-specific routines
SYS$OPDRIVER.EXE	Console terminal port driver
SYS$PUBLIC_VECTORS.EXE	System service vector base image
SYS$*xx*BTDRIVER.EXE	One or more system device boot drivers
VAXVMSSYS.PAR and other parameter files	System configuration parameters

Table 33–1 (Cont.) Files and Programs Used During System Initialization

Files Accessed	Function of File
	EXEC_INIT
CPULOA.EXE	Tables of CPU data
EXCEPTION.EXE	Exception service routines and system services, bugcheck routines
F11BXQP.EXE	File system support
IMAGE_MANAGEMENT.EXE	Image activation services and routines
IO_ROUTINES.EXE	I/O-related routines and system services
LMF$GROUP_TABLE.EXE	Tables of license data
LOCKING.EXE	Lock management routines and system services
LOGICAL_NAMES.EXE	Logical name routines and system services
MESSAGE_ROUTINES.EXE	Message routines and system services
PROCESS_ MANAGEMENT.EXE	Scheduling routines, process creation and control system services
SECURITY.EXE	Security-related routines and system services
SHELL*xx*K.EXE	Process shell, where *xx* is 8, 16, 32, or 64
SYS$NETWORK_ SERVICES.EXE	DECnet support
SYS$TTDRIVER.EXE	Terminal class driver
SYS$VM.EXE	Page fault service and related routines, virtual address space system service routines, swapper and supporting routines, and related system services
SYS$*xx*DRIVER.EXE	One or more run-time drivers for the system device
SYSDEVICE.EXE	Pseudo device drivers and mailbox system services
SYSGETSYI.EXE	$GETSYI system service
SYSLICENSE.EXE	$LICENSE system service
	SYSINIT Process
DDIF$RMS_EXTENSION.EXE	Support for Digital Document Interchange Format (DDIF) file operations
RECOVERY_UNIT_ SERVICES.EXE	Record Management Services (RMS) recovery services
RMS.EXE	RMS
SYSLDR_DYN.EXE	Dynamic executive image loader
SYSMSG.EXE	System message file

Table 33–1 (Cont.) Files and Programs Used During System Initialization

Files Accessed	Function of File
Startup Process	
DCL.EXE	CLI, mapped into P1 space to interpret and execute commands
DCLTABLES.EXE	Command tables, mapped into P1 space and used by DCL.EXE
LOGINOUT.EXE	First image that runs in startup process
STARTUP.COM	SYS$INPUT for startup process
PAGEFILE.SYS	System page file
SWAPFILE.SYS	System swap file
SYCONFIG.COM	Site-specific device configuration command procedure
SYLOGICALS.COM	Site-specific logical names
SYPAGSWPFILES.COM	Site-specific page and swap files
SYSTARTUP_V5.COM	Site-specific startup command procedure
VMS$LAYERED.DAT	Procedure definition data file for layered products
VMS$PHASES.DAT	Startup phase definition data file
VMS$VMS.DAT	Procedure definition data file for VMS
Various	Procedures and images defined by previous two data files
Install Utility, in Context of Startup Process	
VMSIMAGES.DAT	List of images to be installed
All installed images	Set up as known images
SYSGEN Utility, in Context of Startup Process	
VAXVMSSYS.PAR	SYSGEN parameters
IOGEN Utility, in Context of Startup Process	
Various device drivers	I/O database and device initialization

33.2 Console Subsystem

The preliminary steps in the initialization of a VMS system depend on the particular [*censored*] system being booted. The console subsystem, or simply, the console, is the part of the system that assumes control when the system is powered on. It performs the following tasks:

- It initializes, tests, and prepares the system for system software.
- It loads system software into memory and transfers control to it.
- It provides services to system software that simplify control of and access to the system hardware.

- It provides a means for an operator to monitor and control the system hardware.

This section describes aspects of the console that are relevant to a basic understanding of [*censored*] VMS system initialization.

The basic elements of a [*censored*] console subsystem include a processor, software to run on it, storage for the software, and a terminal.

The console processor runs console software to implement console functionality. There is no requirement that this be a separate and distinct processor. In fact, on most [*censored*] systems the console processor is also the [*censored*] processor.

Console software gives the console operator access to all required console functionality at all times, even when console software runs on a console processor that is also the [*censored*] processor.

[*Censored*] VMS Version 1.0 does not support multiprocessing. However, the [*censored*] console architecture provides for multiprocessing. In a multiprocessing system, there is only one primary processor and one or more secondary processors. Any processor that can access the console terminal, console I/O devices, and the system's battery-backed watch is capable of being the primary processor. The mechanism by which the primary processor is chosen depends on the implementation. Only the primary processor can be the console processor. Secondary processors communicate with the console terminal through the primary processor.

Console storage is an area from which console software is loaded. Typically, it is nonvolatile read-only memory (NVROM). Unlike many VAX consoles, [*censored*] consoles do not generally have secondary storage such as disk or tape. For this reason, [*censored*] consoles do not generally support console command procedures. (Note that remote software that behaves like a console operator by connecting to the console terminal can execute a series of console commands to achieve the effect of a console command procedure.)

The console operator is a human operator or remote software that can monitor and control the console's actions through the console terminal. Console software displays messages to and accepts commands from the console operator through the console terminal. VMS uses the console terminal to display messages during system initialization, during a crash, and at other times, for example, to print security, error logging, and operator request messages. Section 33.2.5.3 provides information on console terminal routines through which VMS accesses the console terminal.

Console software interacts with VMS through the following:

- Environment variables, a few of which allow the console operator to control some of the console's actions and communicate information about the booting environment to VMS (see Section 33.2.3)
- Console callback routines, which allow VMS to access console functionality directly (see Section 33.2.4)
- The hardware restart parameter block (HWRPB), which contains information shared between console software and VMS (see Section 33.2.5)

PALcode, briefly described in Chapter *System Overview*, is part of console software. Console software typically loads PALcode from console storage into system memory prior to system bootstrap, as outlined in Section 33.2.2.

Upon power up, console software locates and initializes system hardware blocks. Next, it examines the environment variable AUTO_ACTION (see Section 33.2.3). If AUTO_ACTION is set to BOOT or RESTART, console software initiates a cold bootstrap sequence, described in Section 33.2.2. If AUTO_ACTION is set to HALT, console software enters the halt state, displaying the following prompt at the console terminal and awaiting commands from the console operator: >>>.

In response to a BOOT command, described in Section 33.2.1, console software initiates a bootstrap.

33.2.1 BOOT Console Command

Console software attempts to locate, load and transfer the primary bootstrap program from the boot devices specified in the BOOT console command, which has the following form:

```
>>>BOOT [[-FLAGS SYSTEM_ROOT,BOOT_FLAGS] [DEVICE_LIST]]
```

The qualifier -FLAGS indicates that the next two comma-separated strings are the parameters SYSTEM_ROOT and BOOT_FLAGS. Console software passes both parameters to APB without interpretation as an ASCII string like 0,0 in the environment variable BOOTED_OSFLAGS.

The parameter SYSTEM_ROOT specifies the hexadecimal number of the root directory on the system disk device in which the bootstrap files and programs listed in Table 33–1 reside. A root directory is a top-level directory whose name is of the form SYS*nn*, where *nn* is the number specified by SYSTEM_ROOT.

The parameter BOOT_FLAGS specifies the hexadecimal representation of the sum of the desired boot flags. Table 33–2 lists possible boot flags and their values. VMS defines names for these flags in the $SWRPBDEF macro. For simplicity, the names are shown in the table without the prefix SWRPB_BOOT_FLAGS$V_.

The parameter DEVICE_LIST is a list of device names, delimited by commas, from which the console must attempt to boot. A device name in DEVICE_LIST does not necessarily correspond to the VMS device name for a given device. In fact, console software translates the device name to a path name before it attempts bootstrap. The path name enables the console to locate the boot device through intervening adapters, buses, and widgets. The path name specification and the algorithm that translates the device name to a path name are system-specific.

During console initialization, console software translates the device list in the BOOTCMD_DEV environment variable (see Section 33.2.3) into path names and records the translated device list in the BOOT_DEV environment variable. If the DEVICE_LIST argument of the BOOT command is empty, console software uses the value of BOOT_DEV as the boot device list.

Table 33–2 Boot Flags

Hexadecimal Value	Name	Meaning if Set
1	CONV	Bootstrap conversationally, that is, allow the console operator to modify SYSGEN parameters in SYSBOOT
2	DEBUG	Map XDELTA to running system
4	INIBPT	Stop at initial system breakpoint
8	DIAG	Perform diagnostic bootstrap
10	BOOBPT	Stop at bootstrap breakpoints
20	NOHEADER	Secondary bootstrap image contains no header
40	NOTEST	Inhibit memory test
80	SOLICIT	Prompt for the name of the secondary bootstrap file
100	HALT	Halt before secondary bootstrap
200	SHADOW	Reserved
400	ISL	Reserved
800	BOOTLOG	Reserved
1000	DEBUG_BOOT	Reserved
2000	CRDFAIL	Mark corrected read data error pages bad
4000	NO_SLICE	Reserved
8000	BIT15	Reserved
10000	DBG_INIT	Enable verbose mode in APB, SYSBOOT, and EXEC_INIT

In response to the BOOT console command, console software attempts to boot from devices in the boot device list, starting with the first one. As it attempts to boot from a specific device, console software initializes the BOOTED_DEV environment variable with the path name of that device. If an attempt to boot from a specific device fails, console software attempts to boot from the next device in the list. If all attempts fail, console software prints an error message on the console and enters the halt state to await operator action.

Later, APB uses the value in BOOTED_DEV to determine the boot device.

33.2.2 Initialization Performed by Console Software

Before transferring control to the primary bootstrap program, console software performs either a cold or a warm bootstrap, depending on whether the system must be fully or partly initialized. Typically, console software does this in response to a BOOT console command, an operating system reboot request, or an error halt that occurs when the AUTO_ACTION environment variable has been set to BOOT.

By default, console software always attempts a warm bootstrap unless a cold bootstrap is implicitly (through the BOOT_RESET environment variable) or explicitly requested. A warm bootstrap is possible only if a previous cold bootstrap has already initialized the hardware blocks and the console software data structures, in particular, the HWRPB. Console software verifies this by checking the consistency of the HWRPB. Performing a warm bootstrap is faster and more efficient because console software can skip certain initial steps, such as initializing hardware blocks and building console software data structures, that it must perform during a cold bootstrap.

Console software always performs a cold bootstrap in response to the following:

- The first BOOT command after the system is powered up
- The INITIALIZE console command when the AUTO_ACTION environment variable has been set to BOOT (see Section 33.2.3)
- The BOOT console command or an error halt condition when the BOOT_RESET environment variable has been set to ON (see Section 33.2.3)
- A request from the VMS executive to perform a cold reboot (see Section 33.2.5.1)

The following list summarizes console actions for a cold bootstrap. Note that all console implementations do not necessarily perform the steps in this order.

1. Console software locates and initializes CPU, memory, and I/O modules.
2. It determines the amount of physical memory available on the system and tests a sufficient amount for the following components:
   - — PALcode memory and scratch areas
   - — Console software data structures and routines that are shared by VMS
   - — Page tables for bootstrap virtual address space
   - — The primary bootstrap program
   - — The initial boot stack
3. It builds the HWRPB and its associated substructures.
4. Console software loads PALcode from console storage into memory. It records the physical addresses and sizes of the PALcode memory and scratch areas in the appropriate per-CPU slot of the HWRPB (see Section 33.2.5.1). It initializes SLOT$IQ_STATE to indicate that PALcode memory is valid and that PALcode is loaded and valid (see Table 33–7).
5. It creates level 1, 2, and 3 page tables, shown in Figure 33–2, for bootstrap virtual address space. The layout of bootstrap virtual address space at the time the primary bootstrap program gains control is shown in Figure 33–3. Chapter *Memory Management Data Structures* describes page tables and memory management.

   Note that console software does not map the level 1 page table (L1PT) in bootstrap address space. (Later, SYSBOOT maps the L1PT in system space.)

Figure 33–2 Bootstrap Page Tables

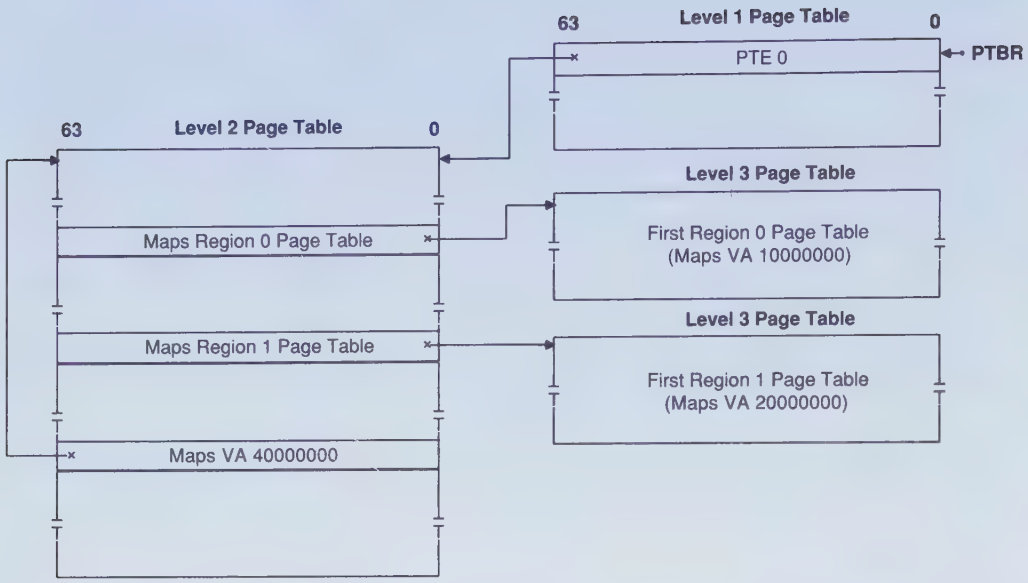
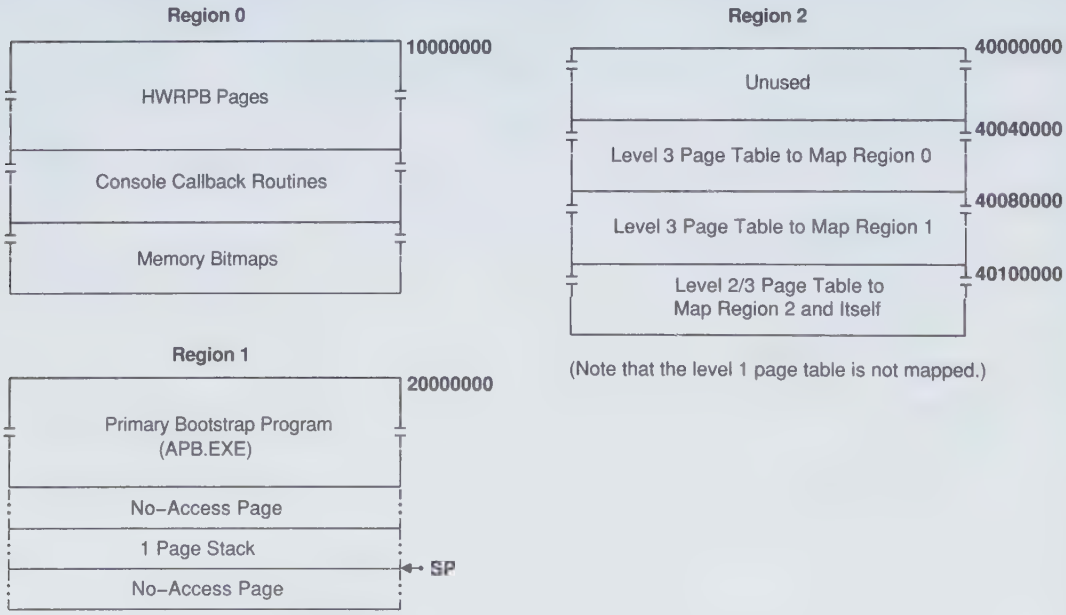


Figure 33–3 Bootstrap Virtual Address Space after APB Is Loaded



(Note that the level 1 page table is not mapped.)

Table 33–3 Processor State Immediately Prior to Primary Bootstrap

Processor Register	Initialized State
Address space number (ASN)	0
AST enable (ASTEN)	0 (all disabled)
AST summary (ASTSR)	0
Floating enable (FEN)	0 (disabled)
Interrupt priority level (IPL)	31
Machine check error summary (MCES)	0
Privileged context block base (PCBB)	200 * (CPU ID) + (contents of HWRPB$IQ_SLOT_OFFSET)
Processor status (PS)	IPL 31, all else is 0
Page table base (PTBR)	PFN of L1PT
Software interrupt summary (SISR)	0
Who am I (WHAMI)	Processor number
System cycle counter (SCC)	0
Processor cycle counter (PCC)	0
Stack pointer (SP)	20000000_{16} + (size of APB.EXE) + 2 * (page size)
Kernel stack pointer (KSP)	Same as SP
Other internal processor registers	Unpredictable
Other integer registers	Unknown
Floating-point registers	Unknown

6. It maps the HWRPB and console callback routines into bootstrap address space beginning at virtual address 10000000_{16}.
7. It locates and loads the primary bootstrap program, as described in Section 33.2.6.
8. Console software builds a hardware privileged context block (HWPCB) in the per-CPU slot (SLOT) for the primary processor (see Section 33.2.5.2). It initializes HWPCB$IQ_KSP with the high-end address of the bootstrap stack and HWPCB$IQ_PTBR with the page frame number (PFN) of the L1PT. It then sets SLOT$V_CV in SLOT$IQ_STATE to indicate that the HWPCB is valid.
9. It initializes every processor to the state shown in Table 33–3.
10. It transfers control to the first longword of the primary bootstrap program in kernel mode at IPL 31 with memory management enabled.

33.2.3 Environment Variables

The console communicates information through environment variables to the console operator and VMS. Also, a console operator or VMS can control some of the console's actions by setting certain environment variables. A console operator accesses environment variables through console commands; VMS accesses environment variables through console callback routines (see Section 33.2.5.4).

An environment variable consists of an identifier and an associated ASCII string. The identifier is a numeric value between 0 and FF_{16}. The *censored* architecture divides environment variables into three categories based on identifier value, as shown in Table 33–4.

Table 33–5 summarizes environment variables common to all system implementations. A console operator identifies an environment variable by its name. VMS uses the numeric identifiers to access environment variables (see Section 33.2.5.4).

Table 33–4 Numerical Ranges for Environment Variables

Range	Description
$0–3F_{16}$	Common to all system implementations
$40_{16}–7F_{16}$	Specific to a system implementation
$80_{16}–FF_{16}$	Specific to system software

Table 33–5 Standard Environment Variables

ID	Symbol	Key	Description
1	AUTO_ACTION[1]	NV, W	Console action following power on or an error halt. Set to BOOT, HALT, or RESTART.
2	BOOT_DEV	W	Default bootstrap device list. The console derives the value from BOOTCMD_DEV at console initialization; the value is preserved across warm bootstraps.
3	BOOTCMD_DEV[1]	NV, W	Default bootstrap device list for BOOT command.

[1]Although console software uses these variables, VMS does not currently access them. Note that VMS does provide system services to access environment variables (see Chapter *Miscellaneous System Services*).

NV—Nonvolatile. The last value saved by system software or set by console commands is preserved across system initializations, cold bootstraps, and long power outages.

W—Warm nonvolatile. The last value set by system software is preserved across warm bootstraps and restarts.

RO—Read-only. The variable cannot be modified by system software or console commands.

Table 33–5 (Cont.) Standard Environment Variables

ID	Symbol	Key	Description
4	BOOTED_DEV	RO	Specific device used by most recent or in-progress bootstrap attempt; see BOOT_DEV.
5	BOOT_FILE[1]	NV, W	Default bootstrap file name.
6	BOOTED_FILE[1]	RO	Bootstrap file name used by most recent or in-progress bootstrap attempt.
7	BOOT_OSFLAGS	NV, W	Default value for the –FLAGS BOOT command qualifier to be passed to APB.
8	BOOTED_OSFLAGS	RO	Value of the –FLAGS BOOT command qualifier during most recent or in-progress bootstrap attempt.
9	BOOT_RESET[1]	NV, W	When set to ON, indicates that a cold bootstrap must be performed in response to an error halt or BOOT command.
A_{16}	DUMP_DEV[1]	NV, W	Crash dump device.
B_{16}	ENABLE_AUDIT[1]	NV, W	When set to ON, enables display of audit trail messages during bootstrap.
C_{16}	LICENSE[1]	NV, RO	Software license in effect. MU for multiple user system and SU for single user system.
D_{16}	CHAR_SET[1]	NV, W	Current console terminal character-set encoding.
E_{16}	LANGUAGE[1]	NV, W	Current console terminal language.
F_{16}	TTY_DEV	NV, W, RO	Current console terminal unit.

[1]Although console software uses these variables, VMS does not currently access them. Note that VMS does provide system services to access environment variables (see Chapter *Miscellaneous System Services*).

NV—Nonvolatile. The last value saved by system software or set by console commands is preserved across system initializations, cold bootstraps, and long power outages.

W—Warm nonvolatile. The last value set by system software is preserved across warm bootstraps and restarts.

RO—Read-only. The variable cannot be modified by system software or console commands.

33.2.4 Console Callback Routines

The term *callback routine* describes a routine in console software that the operating system can call. Console software provides VMS with a set of callback routines to perform I/O operations to console subsystem devices, to access environment variables, and to perform address fixups (see Section 33.2.5.5) within the callback routines if they are remapped by VMS. (Console callback routines are in bootstrap address space; after VMS creates system virtual address space,

it deletes bootstrap address space. To continue to use console callback routines, VMS must remap them into system address space before it deletes bootstrap address space.)

A console callback routine can be called only in kernel mode and only with the HWRPB and all console callback routines virtually mapped. Table 33–6 lists common console callback routines.

Some console callback routines modify their operating context and can be called only during system bootstrap or crash, when the operating system is not fully functional. Other console callback routines do not modify their operating context and can be called in any environment. VMS provides a system service interface to the SET_ENV and GET_ENV callback routines, enabling environment variables to be accessed under program control (see Chapter *Miscellaneous System Services*).

VMS performs all console terminal I/O through the console terminal callback routines. VMS does not use the generic I/O callback routines. All console callback routines but FIXUP are indirectly entered through the DISPATCH callback routine. Section 33.2.5.4 describes how VMS locates and calls console callback routines.

33.2.5 Hardware Restart Parameter Block

The HWRPB contains information that is shared between the console and VMS. Console software creates the HWRPB in physically contiguous memory during console initialization and uses it during and after bootstrapping. Before passing control to APB, console software maps the HWRPB in bootstrap space at virtual address 10000000_{16}. During system initialization, SYSBOOT remaps the HWRPB into system virtual address space and records its address in global location EXE$GPQ_HWRPB.

Figure 33–4 shows the HWRPB. Significant fields of the HWRPB are briefly described here. The HWRPB has a number of substructures, which are described in subsequent sections.

HWRPB$PQ_BASE contains the physical address of the HWRPB. HWRPB$IQ_IDENT contains the null-terminated ASCII string HWRPB. Both fields are used by the console to validate the HWRPB. HWRPB$IQ_REVISION identifies the format of the HWRPB. HWRPB$IQ_SIZE contains the size in bytes of the HWRPB.

HWRPB$IQ_PRIMARY contains the processor identification (ID) of the primary processor in a multiprocessing system. Although [*censored*] VMS Version 1.0 does not support multiprocessing, the console subsystem architecture provides support for multiprocessing.

Table 33–6 Common Console Callback Routines

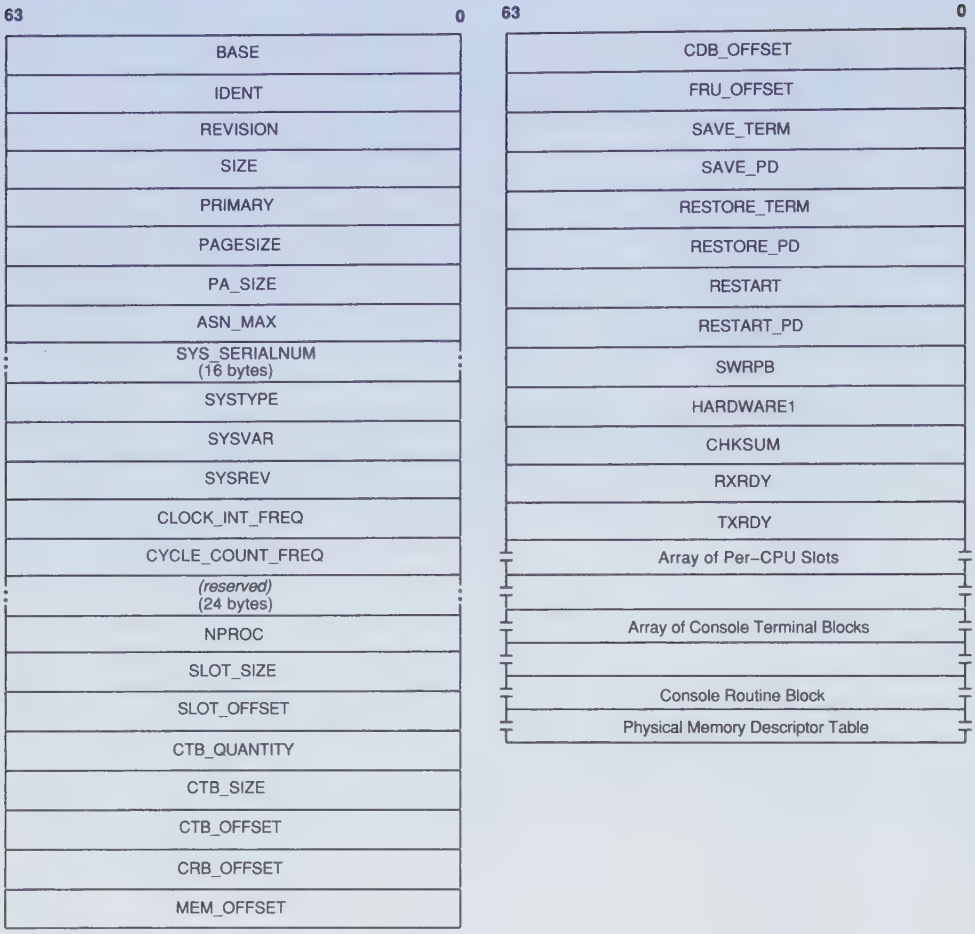
Value	Name	Description
	Console Terminal Routines	
1	GETC	Get character from console terminal
2	PUTS	Put byte stream to console terminal
3	RESET_TERM	Reset console terminal to default
4	SET_TERM_INT	Enable or disable console terminal interrupts
5	SET_TERM_CTL	Enable or disable console terminal controls
6	PROCESS_KEYCODE	Process and translate keycode
$7–F_{16}$		Reserved
	Console Generic Device I/O Routines	
10_{16}	OPEN[1]	Open I/O device for access
11_{16}	CLOSE[1]	Close I/O device for access
12_{16}	IOCTL[1]	Perform I/O device-specific operation
13_{16}	READ[1]	Read I/O device
14_{16}	WRITE[1]	Write I/O device
$15_{16}–1F_{16}$		Reserved
	Console Environment Variable Routines	
20_{16}	SET_ENV	Write an environment variable
21_{16}	RESET_ENV	Reset an environment variable to its default
22_{16}	GET_ENV	Read an environment variable
23_{16}	SAVE_ENV[2]	Save current environment variable
	Console Miscellaneous Routines	
None	FIXUP	Remap console callback routines
None	DISPATCH	Dispatch to console callback routine

[1]VMS does not use this routine.

[2]Most [*censored*] console subsystems do not provide this routine.

HWRPB$IQ_PAGESIZE contains the number of bytes within a page for this processor. HWRPB$IQ_PA_SIZE contains the number of bits needed to represent the highest physical address for the system. The [*censored*] architecture restricts this number to 48 or less. HWRPB$IQ_ASN_MAX contains the maximum address space number value allowed by this processor. Chapter *Memory Management Overview* describes [*censored*] physical and virtual address representations.

Figure 33–4 Hardware Restart Parameter Block



(continued)

HWRPB$B_SYS_SERIALNUM contains a ten-byte ASCII serial number stored in the system at the time of manufacture. HWRPB$IQ_SYSTYPE contains the system type, and HWRPB$IQ_SYSVAR, the system subtype. HWRPB$IQ_SYSREV contains a four-byte ASCII string that indicates system revision information.

HWRPB$IQ_CLOCK_INT_FREQ contains 4,096 times the number of interval timer interrupts per second. HWRPB$IQ_CYCLE_COUNT_FREQ contains the number of times per second the system cycle counter and process cycle counter are updated. Chapter *Time Support* describes interval timer interrupts and these counters.

HWRPB$IQ_CRB_OFFSET contains the offset to the HWRPB_CRB, the console routine block (see Section 33.2.5.4); and HWRPB$IQ_MEM_OFFSET contains the offset to the HWRPB_PMD, the physical memory descriptor table (see Section 33.2.5.5).

After VMS initialization, HWRPB$IQ_RESTART will contain the virtual address of the code entry point of EXE$RESTART, the processor restart routine. HWRPB$IQ_RESTART_PD will contain the virtual address of its procedure descriptor. Together, these two fields form a linkage pair. Chapter *Power Failure and Recovery* describes EXE$RESTART and the circumstances under which it is entered.

HWRPB$PQ_SWRPB contains the physical address of the software restart parameter block (SWRPB), which is described in Section 33.3.1.

HWRPB$IQ_CHKSUM contains the checksum of all quadwords from the beginning of the HWRPB up to and including the quadword at HWRPB$IQ_HARDWARE1. The checksum is computed as a 64-bit 2's complement sum ignoring overflows. It is used to validate the HWRPB during warm bootstraps and restarts. It is set during console initialization and recomputed each time the console or VMS modifies any field in the HWRPB within the region for which the checksum is computed.

33.2.5.1 Per-CPU Slot

Information about the state of a processor is contained in the SLOT for that processor.

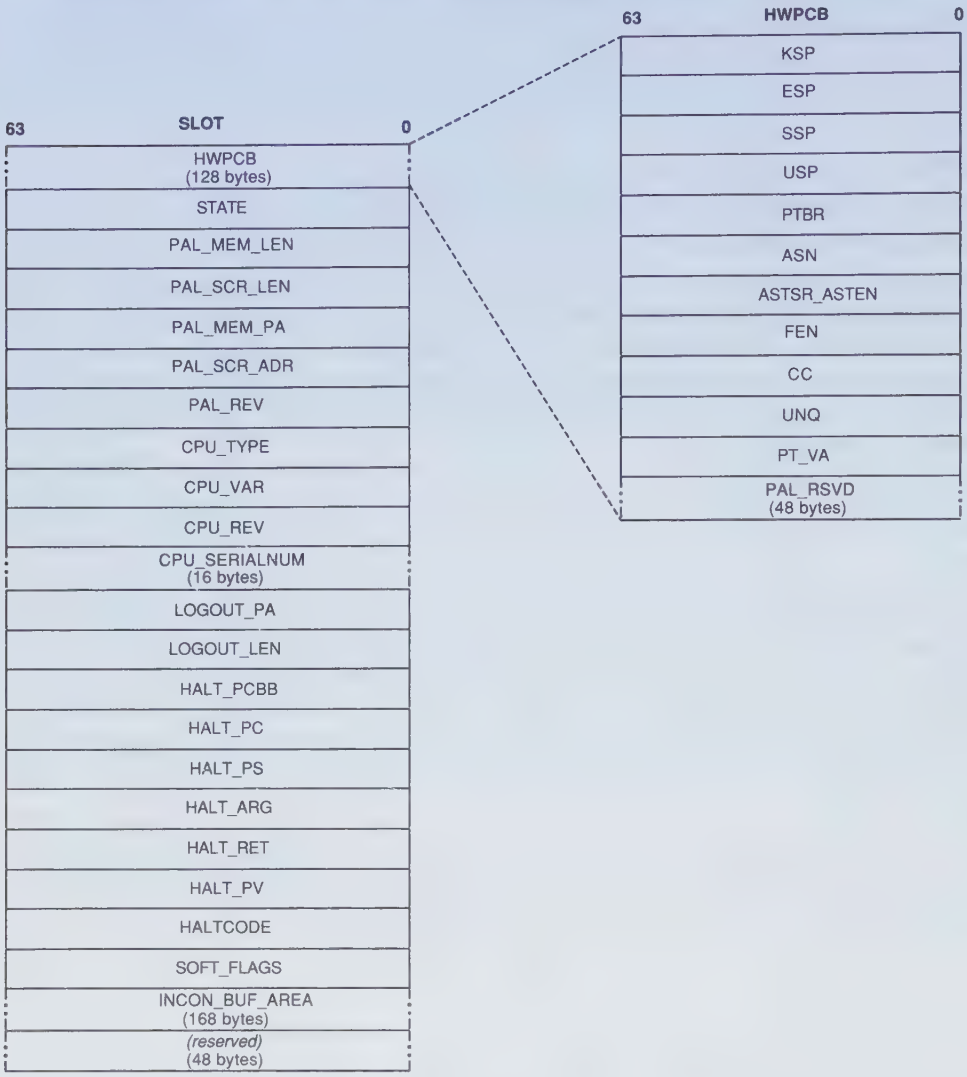
The HWRPB has an array of n SLOTs, where n is the number of processors and $n - 1$ is the highest processor ID on the system. HWRPB$IQ_SLOT_OFFSET contains the offset from the beginning of the HWRPB to the base of this array. HWRPB$IQ_NPROC contains the number of SLOTs present in the HWRPB. HWRPB$IQ_SLOT_SIZE contains the size in bytes of each SLOT rounded up to the next multiple of 128. (Note that [*censored*] VMS Version 1.0 does not support multiprocessing.)

Each SLOT contains a HWPCB substructure for that processor. Figure 33–5 shows the SLOT and the HWPCB. Console software initializes the primary processor's HWPCB with the limited context of the bootstrap HWPCB. Section 33.2.5.2 discusses the bootstrap and other HWPCBs.

In addition, a SLOT contains the following:

- Processor state flags in SLOT$IQ_STATE, listed in Table 33–7.
- Processor halt action mask, also in SLOT$IQ_STATE, whose possible values are listed in Table 33–8. Prior to requesting a halt, VMS records one of these values in SLOT$IQ_STATE. When VMS executes a HALT instruction, console software examines this location and takes appropriate action.
- Fields that describe PALcode memory space, scratch space, and revision.
- Processor type, variation, revision, and serial number.
- The values of various processor registers at the last time the processor was halted.

Figure 33–5 Per-CPU Slot and Hardware Privileged Context Block



33.2.5.2 Hardware Privileged Context Block

A process's hardware context includes, among others, the following registers: the kernel, executive, supervisor, and user mode stack pointer registers, the PTBR, the ASN register, the AST summary and AST enable registers, the FEN register, and the PCC register. A noncurrent process's hardware context is stored in the process's HWPCB.

The *censored* architecture requires that there always be a valid HWPCB to define a context for CPU execution, minimally to specify the address of a kernel stack and the PFN of the L1PT. During bootstrap, the console subsystem builds a HWPCB in the per-CPU slot in the HWRPB. APB and SYSBOOT run in the context of that HWPCB. After a power failure recovery or error halt, the console subsystem transfers control to VMS restart code in the context of that HWPCB.

During system initialization, VMS builds a HWPCB and allocates a kernel stack to define a context in which the system can run when there is no computable process to be placed into execution.

VMS builds another HWPCB that defines a context to which the system can switch in the final stages of saving state during a power failure. VMS also runs in that context after having been restarted. (After restart, it switches context from the boot HWPCB as soon as possible so that the console can perform another restart if necessary.)

The HWPCB built by the console subsystem is often referred to as the boot process; the two HWPCBs built by VMS are often referred to as the system process and the termination process. None, however, represents a process that can be scheduled; each merely provides a limited environment for the execution of kernel mode code in system space at elevated IPL. To avoid confusion of these contexts with normal processes, this book refers to them as the boot HWPCB, system HWPCB, and termination HWPCB.

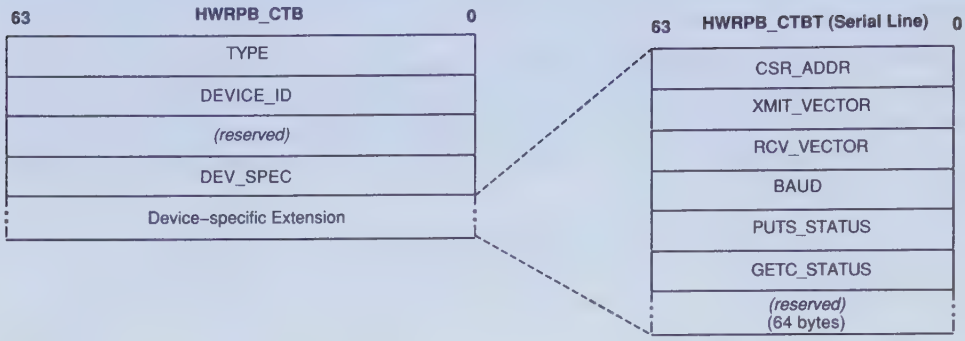
Table 33–7 Processor State Flags

Bit	Meaning if Set
SLOT$V_BIP	Bootstrap in progress
SLOT$V_RC	Processor is restart capable
SLOT$V_PA	Processor is available
SLOT$V_PP	Processor is present
SLOT$V_OH	Processor has been halted by explicit operator action
SLOT$V_CV	Processor's HWPCB is valid
SLOT$V_PV	PALcode is valid
SLOT$V_PMV	PALcode memory is valid
SLOT$V_PL	PALcode has been loaded

Table 33–8 Processor Halt Action Mask

Mask	Action Taken when Processor Halts
SLOT$M_NOACTION	None
SLOT$M_SAVE_RESTORE_TERM	Save or restore console terminal
SLOT$M_COLD_REBOOT	Initiate cold bootstrap
SLOT$M_WARM_REBOOT	Initiate warm bootstrap
SLOT$M_REMAIN_HALTED	Do not attempt restart

Figure 33–6 Console Terminal Block



33.2.5.3 Console Terminal Block

A console terminal block (HWRPB_CTB) describes a terminal device connected to the console. There can be one or more such devices; however, only one of them is designated as the console terminal. Console software communicates information about console terminal I/O and interrupts to the VMS console terminal driver (OPDRIVER) through the HWRPB_CTB.

The HWRPB contains a table of HWRPB_CTBs (see Figure 33–4). HWRPB$IQ_CTB_QUANTITY contains the number of HWRPB_CTBs in the table. HWRPB$IQ_CTB_SIZE contains the size of the largest HWRPB_CTB in the table. HWRPB$IQ_CTB_OFFSET contains the offset to the table from the HWRPB base. The index of a HWRPB_CTB in the table is the unit number of the associated terminal device.

A HWRPB_CTB consists of a device-independent portion and a device-specific portion. Figure 33–6 shows the HWRPB_CTB and the device-specific extension for a serial line interface. Each HWRPB_CTB contains its own length in HWRPB_CTB$IQ_DEV_SPEC.

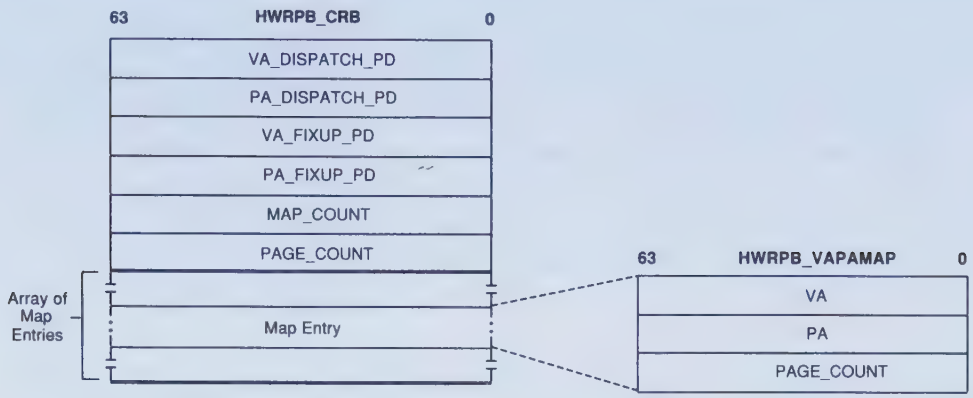
During console initialization, the console creates HWRPB_CTBs for all terminal devices attached to the console, records their default states in the respective HWRPB_CTBs, and stores the unit number of the designated console terminal in the environment variable TTY_DEV.

33.2.5.4 Console Routine Block

The HWRPB_CRB, shown in Figure 33–7, enables VMS to locate the DISPATCH and FIXUP routines. Additionally, it contains information for mapping console callback routines and I/O space locations that they reference into virtual address space. If VMS remaps console callback routines, the I/O space locations that they reference, or the HWRPB, it must call FIXUP to adjust virtual address references in the console callback routines.

To call a particular console callback routine, VMS calls DISPATCH, passing as its first argument the corresponding numeric value listed in Table 33–6. VMS defines symbolic names for these numeric values in the $HWRPBDEF macro. The remaining arguments to DISPATCH depend on the function requested. The

Figure 33–7 Console Routine Block



following BLISS code fragment illustrates how the DISPATCH routine is called to get the value of an environment variable:

```
!Get address of DISPATCH routine from console routine block
!
BIND ROUTINE DISPATCH = .crb[hwrpb_crb$iq_va_dispatch_pd];
status = DISPATCH( hwrpb_crb$k_getenv,      !Get environment variable
                   hwrpb_crb$k_booted_dev,  !BOOTED_DEV
                   devnambuf,               !Buffer to get value in
                   16);                     !Length of buffer
```

33.2.5.5 Physical Memory Descriptor Table

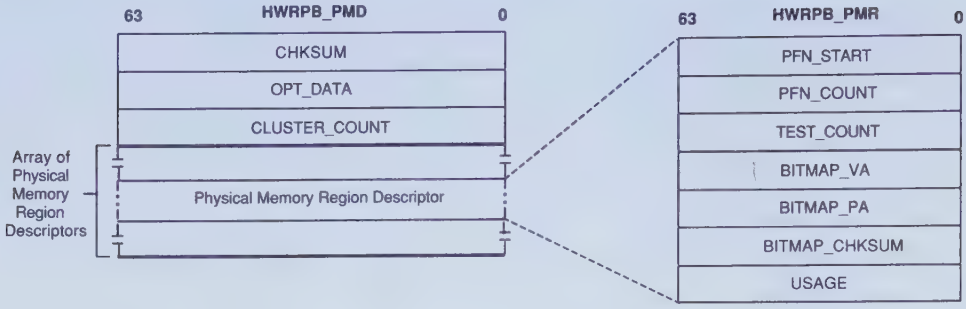
During cold bootstrap, the console sizes physical memory on the system. Typically, it permanently reserves some memory for its own use and to load PALcode. It builds a table, the HWRPB_PMD, to describe memory and identify the reserved pages. VMS uses this table to determine what memory is available for its own use. Both console software and VMS use the table to record information about which pages of memory are unavailable either because they have not yet been tested or because they failed the test.

The HWRPB_PMD has a fixed part, which describes the structure as a whole, and a variable part. The variable part consists of an array of physical memory region descriptors (HWRPB_PMRs). Figure 33–8 shows the HWRPB_PMD and a HWRPB_PMR.

In the fixed part, HWRPB_PMD$IQ_CHKSUM is the checksum of all the remaining quadwords in the HWRPB_PMD. HWRPB_PMD$IQ_OPT_DATA is the physical address of system-specific data; if no such data exists, the console zeros this field. HWRPB_PMD$IQ_CLUSTER_COUNT contains the number of HWRPB_PMRs that immediately follow.

Each HWRPB_PMR describes a memory cluster, a series of physically contiguous pages of memory. If the flag HWRPB_PMR$V_CONSOLE in HWRPB_PMR$IQ_USAGE is clear, the descriptor and the memory cluster it describes are available to system software; otherwise, they are in use by console software.

Figure 33–8 Physical Memory Descriptor Table



HWRPB_PMR$IQ_PFN_START contains the PFN of the first physical page of memory in the cluster. HWRPB_PMR$IQ_PFN_COUNT contains the number of physical pages in the cluster. HWRPB_PMR$IQ_TEST_COUNT contains the number of pages in the cluster that have already been tested.

HWRPB_PMR$IQ_BITMAP_VA and HWRPB_PMR$IQ_BITMAP_PA contain the starting virtual and physical addresses of the cluster's memory testing bitmap in bootstrap address space. (Note that SYSBOOT remaps the bitmap into system space.) If the cluster is untested, or used by the console, these fields are zero. HWRPB_PMR$IQ_BITMAP_CHKSUM contains the checksum of the cluster's memory testing bitmap. In the memory testing bitmap, each bit represents a page of memory. If the bit is set, the page is available for use; otherwise it is not. If the bit is not set, either the page may not have been tested or it may be bad.

A typical console implementation creates two HWRPB_PMRs in the HWRPB: one that describes memory used by the console (including memory used by and for PALcode), and one that describes memory used by VMS.

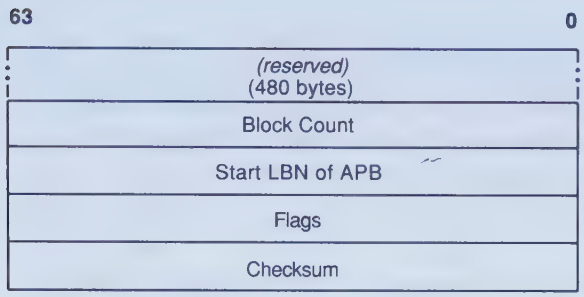
33.2.6 Primary Bootstrap Program Loading

How the primary bootstrap program is located and loaded depends on the type of bootstrap device. [*Censored*] VMS Version 1.0 supports only one type of bootstrap device: a disk.

To bootstrap from a disk, console software reads logical block number (LBN) 0 of the disk. LBN 0 of a bootable disk contains the boot block, shown in Figure 33–9. Note that the first 480 bytes of the boot block typically contain a VAX boot block. This allows a disk to be bootable from either a VAX system or a [*censored*] system, provided that images required for both the VAX VMS and [*censored*] VMS operating systems are present on the disk.

Console software on a [*censored*] system attempts to load the primary bootstrap program using two pieces of information in the boot block: the block count and starting LBN of the primary bootstrap program. This implies that the primary bootstrap program must be present on contiguous disk blocks. Console software has no knowledge of the file system on the disk.

Figure 33–9 Disk Boot Block (LBN 0)



Console software loads the primary bootstrap program into bootstrap virtual address space at address 20000000_{16}.

33.3 [Censored] Primary Bootstrap Program (APB)

APB is the primary bootstrap program for VMS. Running in kernel mode at IPL 31 in the context of the bootstrap HWPCB, it performs the following steps:

1. If necessary, it continues the task of testing system memory. The console subsystem starts this task but may only test enough memory to load APB, depending on console implementation. APB tests enough memory for creating some of its own data structures as well as for loading the secondary bootstrap program.
2. It locates a secondary bootstrap program, loads it into memory, and transfers control to it. By default APB loads SYSBOOT, the VMS secondary bootstrap program. APB can also load other secondary bootstrap programs in response to certain boot flags, as described in Section 33.3.1.

APB and SYSBOOT are conceptually one program. The VAX-11/780 system, on which Version 1.0 of VAX/VMS was implemented, required that the initial bootstrap program reside on the console floppy diskette, whose capacity of 512 blocks was also used for microcode, console software, and console command procedures. Rather than impose artificial restrictions on the size of the bootstrap program, the designers divided the program into two pieces:

- A primary piece that resides in console storage, one of whose major purposes is to locate the secondary piece
- A secondary piece that resides on the system device (with no real limits on its size) that performs the bulk of the bootstrap operation

Once this division was achieved, VMB, the VAX/VMS primary bootstrap program, became a more flexible tool that could load programs other than VMS. To preserve this flexibility, the division of the bootstrap into primary and secondary pieces was continued in subsequent versions of VMS.

Table 33–9 Memory Allocation Cells

Cell	Description
BOO$GQ_PAGE_SIZE[1]	Number of bytes in a page (8K, 16K, 32K, or 64K)
BOO$GQ_PHY_MEM_DESC[1]	Virtual address of physical memory descriptor
BOO$GQ_PTES_PER_PAGE[1]	Number of page table entries (PTEs) to a page
BOO$GQ_BOFF_MASK[1]	Bit mask to extract the byte-within-page field from a PTE
MMG$GL_VA_TO_VPN	Number of bits in the byte-within-page field of a PTE (negated)
BOO$GQ_PTE_SEG_SIZE[1]	Number of bits in each segment of a PTE
BOO$GQ_L1_BITPOS[1]	Bit position of lowest bit in the first segment of a PTE
BOO$GQ_L2_BITPOS[1]	Bit position of lowest bit in the second segment of a PTE
BOO$GQ_L3_BITPOS[1]	Bit position of lowest bit in the third segment of a PTE
BOO$GQ_L1_MASK[1]	Bit mask to extract first segment of a PTE
BOO$GQ_L2_MASK[1]	Bit mask to extract second segment of a PTE
BOO$GQ_L3_MASK[1]	Bit mask to extract third segment of a PTE
BOO$GQ_MAX_PFN[1]	Maximum physical page number on the system
BOO$GQ_HIGH_PFN[1]	Highest tested physical page number

[1]Despite its name, this is a longword cell.

[*Censored*] VMS preserves the separation of APB and SYSBOOT, although the original reasons for such separation no longer apply. Unlike VMB, APB does not load bootstrap programs for other operating systems, although it could be used to load a different secondary bootstrap program for debugging purposes. (Any such use is reserved to Digital and completely unsupported.) The [*censored*] console architecture allows console software to locate the primary bootstrap program for any operating system.

33.3.1 APB Flow

APB runs in kernel mode at IPL 31. Among other things, this blocks interrupts until the mechanisms for handling them are in place. APB performs the following steps:

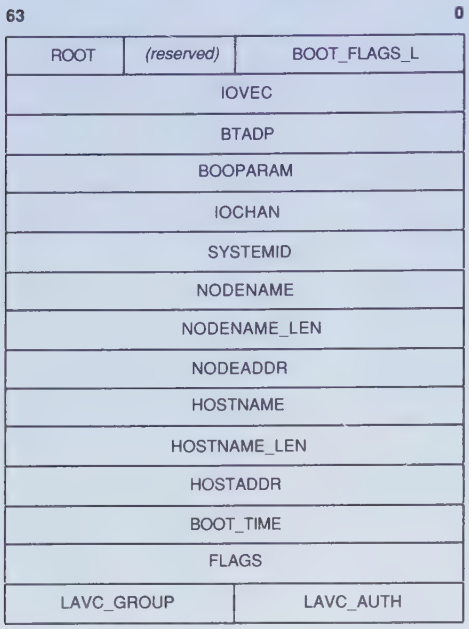
1. It records the system type from HWRPB$IQ_SYSTYPE and the processor type from the per-CPU slot at offset SLOT$IQ_SYSTYPE. It then records the index of the CPU table entry in the CPU dispatch table (see Section 33.3.2.1) that matches the system's and processor's type. APB subsequently uses this index to locate the system-specific routine for a given operation.

Table 33–10 Bootstrap Argument List

Field	Description
ARG_REVISION	Argument list revision
BOOT_REVISION	Primary bootstrap revision
HWRPB	Virtual address of the HWRPB
DISPATCH_VECTOR	Reserved
FILE_CACHE	File cache directory
FILE_CACHE_SIZE	Size of file cache directory
LOW_PFN	Lowest tested PFN
HIGH_PFN	Highest tested PFN
MAX_PFN	Highest available PFN
NEXT_PFN	Next available PFN
PFN_MAP	Address of allocation bitmap
NEXT_VA	Next available virtual address
BOOT_STYLE	Style of booting
IO_CHAN	Boot driver selector
CPU_CHAN	CPU selector
IOVEC	List of boot drivers needed
SYSBOOT_IMAGE_HDR	Virtual address of SYSBOOT's image header
IO_CHAN_MASK	I/O channel mask
IOVEC_SIZE	Size of each entry in IOVEC
IOVEC_CNT	Number of boot drivers needed
FIXUP_BLOCK	Virtual address of fixup block

2. APB initializes a number of cells that are used by the boot time memory management code. These cells, which are initialized based primarily on the system page size in HWRPB$IQ_PAGESIZE, are listed in Table 33–9. APB also builds the memory allocation bitmap through which the allocation of bootstrap virtual address space is managed.
3. APB initializes the primitive console terminal driver that is linked into APB and its associated data structures. Because APB and SYSBOOT run at IPL 31 they must perform polled, rather than interrupt-driven, I/O to the console terminal. APB disables console terminal interrupts.
4. It allocates and initializes the bootstrap argument list (ARG) structure, defined in [APB]BOODEF. APB passes information to the secondary bootstrap program in the ARG. Table 33–10 provides a summary of information in the argument list. For simplicity, the table omits the prefix ARG$IL_ from the offset names for the cells.
5. APB allocates and initializes the SWRPB, shown in Figure 33–10. The SWRPB contains booting-related information that enables VMS to restart the system if necessary. SYSBOOT later copies this structure into nonpaged pool and records its address in EXE$GPQ_SWRPB.

Figure 33–10 Software Restart Parameter Block



6. APB allocates a page of physical memory for the system control block (SCB) and maps the page into bootstrap address space. This is a temporary SCB, used only by APB. The run-time SCB is built by SYSBOOT (see Section 33.4) after it creates the system page table (SPT). APB initializes all entries in the temporary SCB. In each entry, it initializes the SCB vector with the address of an exception handler; it initializes the SCB parameter with its own offset from the SCB base. APB then records the physical address of the SCB in the SCB base register (SCBB). For more information on SCB entries, see Chapter *Interrupts and Exceptions*.

7. APB locates the DISPATCH console callback routine through the HWRPB_CRB and calls it to get the value of the BOOTED_OSFLAGS environment variable. APB parses BOOTED_OSFLAGS, which contains the ASCII representation of the boot flags, and records the binary representation in SWRPB$IQ_BOOT_FLAGS.

   Table 33–2 describes the bits in SWRPB$IQ_BOOT_FLAGS. For simplicity, this chapter omits the prefix SWRPB_BOOT_FLAGS$V_ from their names.

8. It calls the CPU-specific routine to validate PALcode revision information (see Section 33.3.2.1).

9. It calls EXE$INI_TIMWAIT, in module [SYSLOA]TIMEDWAIT, to perform system-specific setup of certain constants required for the proper operation of the TIMEDWAIT macro (see Chapter *Time Support*).

10. APB calls BOO$INIT_XDELTA, in module [APB]EXCEPTION, to initialize the XDELTA debugger. BOO$INIT_XDELTA calls XDT$INIT, in module [DELTA]XDELTA, to set up an SCB entry for the breakpoint fault (BPT) exception and to perform other initialization of the debugging environment.

11. If the BOOBPT boot flag is set, APB executes a CALL_PAL BPT instruction at a location known to XDELTA. As a result of the exception, control transfers to XDELTA. Through XDELTA commands, a console operator can examine and modify locations in APB data structures and set breakpoints in APB to troubleshoot problems that prevent a system from being booted.
12. APB determines the boot device type from the BOOTED_DEV environment variable. The returned value corresponds to one of the values defined in module [APB]BDTDEF.
13. It selects one or more boot drivers for the boot device and calls the unit initialization routine in each selected driver (see Section 33.3.2.2).
14. It initializes some remaining parts of the bootstrap argument list.
15. APB selects the secondary bootstrap file, which must reside on an On-Disk Structure Level 2 (ODS-2) Files-11 volume, as follows:
    - By default it uses the name [SYSEXE]SYSBOOT.EXE.
    - If the SOLICIT boot flag has been set, it prompts the console operator for the file name of a secondary bootstrap program. Digital uses this flag to debug the secondary bootstrap program. Any other use is completely unsupported.
16. APB mounts the system device and opens the file selected in step 15 as the secondary bootstrap program. It loads the secondary bootstrap program into bootstrap address space, starting at virtual address 0.
17. As its final step, APB transfers control to the secondary bootstrap program. For a secondary bootstrap program like SYSBOOT that contains a standard [*censored*] VMS image header, APB obtains the transfer address from the image header and calls it at its transfer address. APB calls a secondary bootstrap program without a standard VMS image header at the virtual address of 0.

33.3.2 Structure of APB.EXE

Most modules that make up APB.EXE are from the [APB] facility. APB procedures are structured so that APB can be easily extended to support any one of multiple processor types, system types, or boot devices. Note that the XDELTA debugger is linked into APB.EXE. The main components of the APB.EXE image are described in the following sections.

33.3.2.1 CPU Dispatch Table and CPU-Specific Code

For each combination of system and processor type supported by [*censored*] VMS, the system-specific module [APB]CPU*xxyy* provides a set of routines that APB uses. *xx* is the system type designator, and *yy* is the processor type designator.

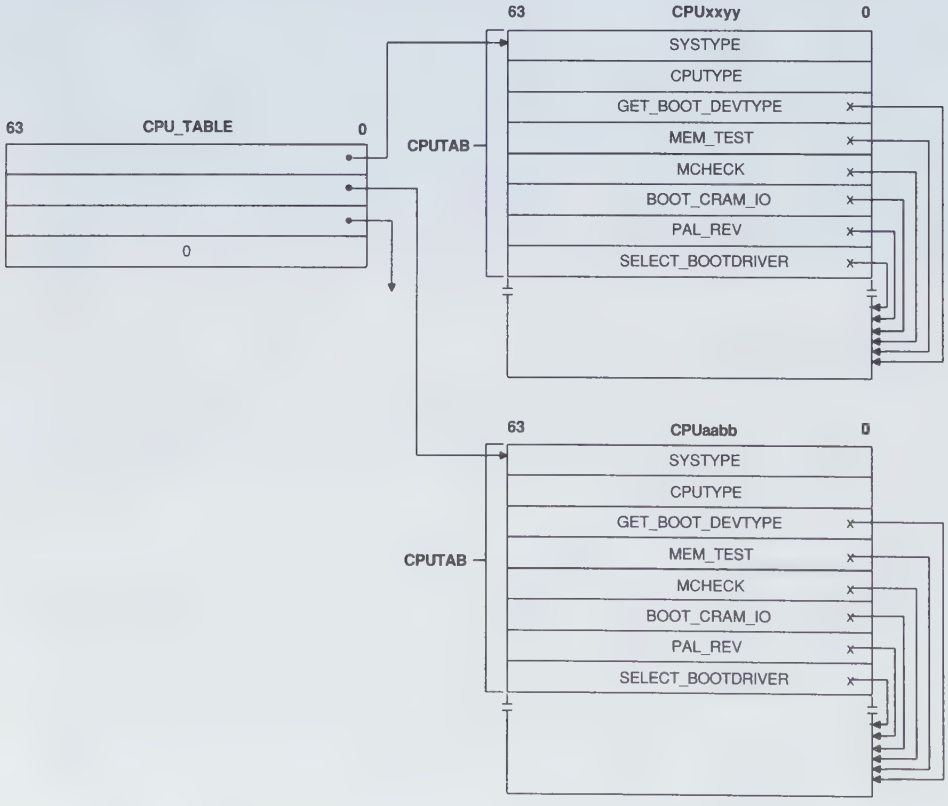
Each system-specific module uses the $CPU macro, defined in [APB]BOOT_MACROS.REQ, to declare its system-specific routines, as shown in this BLISS code fragment in [APB]CPU0302:

```
$CPU(
 SYS_TYPE = hwrpb_systype$k_SYSTEM_C,          !System type
 CPU_TYPE = hwrpb_cpu_type$k_ev3,              !CPU type
 GET_BOOT_DEVTYPE = get_boot_devtype_0302,     !Determine boot device
                                               ! type routine
 MEM_TEST = mem_test_0302,                     !Memory test routine
 MCHECK = mcheck_0302,                         !Machine check routine
 BOOT_CRAM_IO = boot_cram_io_0302,             !CRAM I/O routine
 PAL_REV = boo$null_routine,                   !PAL revision validation
                                               ! routine
 SELECT_BOOTDRIVER = select_bootdriver_0302    !Create IOVEC and select
);                                             ! bootdriver
```

Each invocation of the $CPU macro creates a CPU table (CPUTAB) structure and an entry in the CPU dispatch table (CPU_TABLE). CPU_TABLE and two CPUTAB structures are shown in Figure 33–11. The CPUTAB contains procedure values for the system-specific routines in the module that invoked the $CPU macro. Each entry in CPU_TABLE points to a CPUTAB structure in a system-specific module.

APB's first action is to determine the system and processor types for the system being booted; based on this information, it selects the appropriate system-specific module for use, as described in Section 33.3.1.

Figure 33–11 CPU Dispatch Table and CPUTAB Structures



33.3.2.2 Boot Drivers and the Boot Driver Table

A boot driver is a skeleton device driver that is used only while the system is booting or crashing; it implements the minimum functionality required in such an environment. It performs polled, rather than interrupt-driven, I/O. It bears little resemblance to a fully functional run-time VMS device driver such as one described in Chapter *Overview of the I/O Subsystem*. The only VMS components that use boot drivers are APB, SYSBOOT, EXE$INIT, and VMS bugcheck processing code. Chapter *Error Handling* contains more information on bugcheck processing.

Every [*censored*] VMS boot driver is built in two ways:

- It is linked into APB.EXE.
- It is also linked into a separate executive image called SYS$*xx*BTDRIVER.EXE, where *xx* is the device name designator.

APB and SYSBOOT use the boot drivers built into APB.EXE, which operate in bootstrap virtual address space. EXE$INIT and VMS bugcheck processing code use the boot driver executive images, which operate in system space. After SYSBOOT creates system virtual address space, it loads executive images for the selected boot drivers (see Section 33.4.1).

A boot driver interface consists of three procedures: a unit initialization procedure, a queue I/O request (QIO) procedure, and a unit disconnect procedure. Prior to initial use of the boot driver, VMS calls the unit initialization procedure to establish a logical link to the device. To perform read or write operations to the device, VMS uses the QIO procedure. After it has completed all I/O operations to the device, VMS calls the unit disconnect procedure.

APB is built with a set of boot drivers for all devices from which bootstrapping is supported. Each boot driver invokes the $BOOT_DRIVER macro as shown in the following BLISS code fragment in [APB]DKBTDRIVER:

```
$BOOT_DRIVER (
BOOT_DEVICE = BDT$K_DK,                 !Boot device type
CPU_LIST = (EV3, EV4),                  !CPUs supported by this driver
UNIT_INIT = DK_INIT,                    !Unit init procedure
QIO = DK_QIO,                           !QIO procedure
UNIT_DISC = DK_DISC,                    !Unit disconnect procedure
CLASS_DRIVER_LIST = ('DK',              !System device name
              'SYS$DKDRIVER.EXE'),      !System device driver name
BT_DRIVER_NAME = 'SYS$DKBTDRIVER.EXE', !System boot driver name
DEVICE_CLASS = DC$_DISK,                !Device class
BTADP_SIZE = BTADP$C_LENGTH);           !Size of BTADP structure
```

Each invocation of the $BOOT_DRIVER macro creates a boot driver table (BDTAB) structure, shown in Figure 33–12, and an entry in BOOT_DRIVER_TABLE, an array of pointers to BDTABs for all boot drivers in APB.EXE. Figure 33–13 shows the relation between BOOT_DRIVER_TABLE and two BDTAB structures.

Figure 33–12 BDTAB Structure

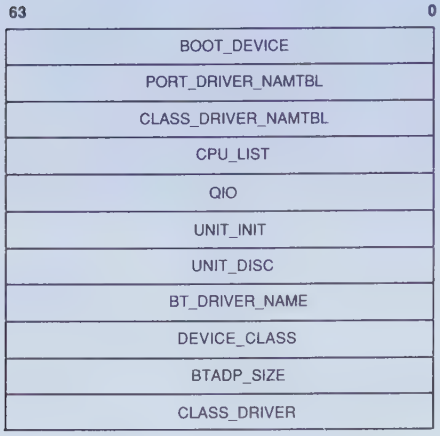
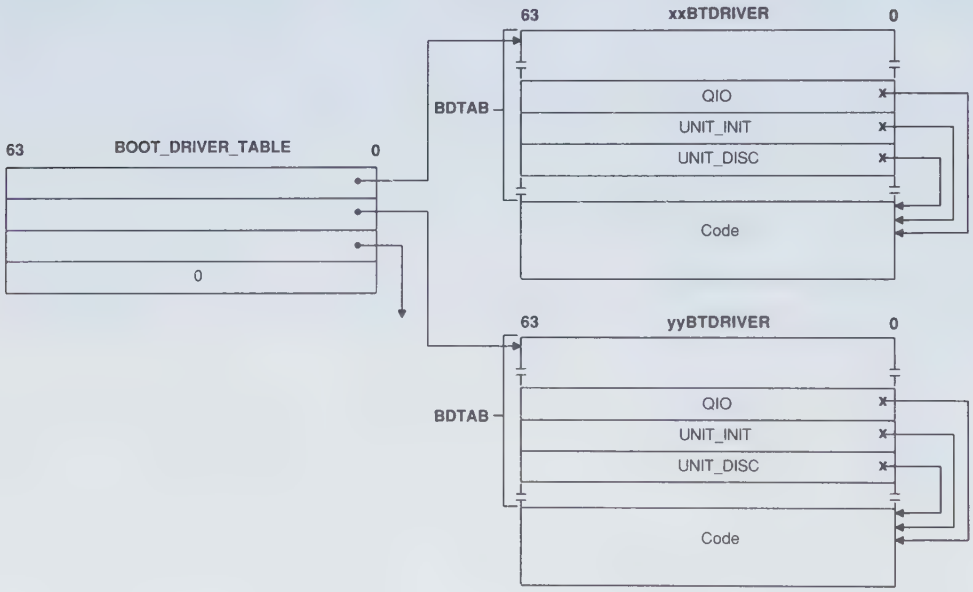


Figure 33–13 Boot Driver Table and BDTAB Structures



33.3.2.3 File System Access Routines

APB.EXE includes two sets of routines for boot time file access. The routines in module [APB]FILE_SYSTEM are independent of file system or device. The routines in module [APB]F11_FILEREAD extend these routines to support primitive file operations for Files-11 ODS-2 files on a supported disk device. Module [APB]FILE_SYSTEM provides the following routines:

- _BOO$MOUNT_VOLUME—Mount boot volume
- _BOO$OPEN_BOOTFILE—Open boot file
- _BOO$LOAD_BOOTFILE—Load boot file

- _BOO$CLOSE_BOOTFILE—Close boot file
- _BOO$DISMOUNT_VOLUME—Dismount boot volume

These routines allow APB to mount the volume that contains the secondary bootstrap program, open the file that contains the image, load the file into memory, close the file, and dismount the volume, independent of the boot device or style of booting.

Booting from a disk, the primary and secondary bootstrap programs must locate files before the file system itself is in full operation. Many files must be looked up before the Files-11 Extended QIO Processor (XQP) has been loaded into memory and initialized by EXE$INIT. Module [APB]F11_FILEREAD exists for this purpose. This module is linked into the APB and SYSBOOT images as well as the executive images PRIMITIVE_IO and EXEC_INIT.

APB and SYSBOOT call _BOO$OPEN_BOOTFILE to locate and read files such as SYS.EXE. _BOO$OPEN_BOOTFILE calls F11FR$OPEN_BOOTFILE, in module [APB]F11_FILEREAD, which calls F11FR$FIND_BOOTFILE, in the same module, to look up the file. To improve performance, F11FR$FIND_BOOTFILE caches information about directories used in the file lookup. For example, locating SYS.EXE might require looking up and reading the master file directory, SYS*n*.DIR, SYSCOMMON.DIR, and SYS$LDR.DIR.

In order to avoid repeated lookups and directory and subdirectory reads, F11FR$FIND_BOOTFILE records directory file IDs, size in blocks, and starting LBN; it also caches blocks from directory files. SYSBOOT copies this primitive file cache to nonpaged pool for use by EXE$INIT and the SYSINIT process until the XQP is operational.

33.4 Secondary Bootstrap Program (SYSBOOT)

SYSBOOT is the VMS secondary bootstrap program. SYSBOOT performs the following major functions:

1. It configures the system by loading a set of adjustable SYSGEN parameters. By default it uses the parameters stored in the file [SYS*n*.SYSEXE]VAXVMSSYS.PAR.

   If a conversational bootstrap was requested, SYSBOOT allows the console operator to change the value of specific SYSGEN parameters, select a whole different set of parameters from a different parameter file, or use a set of default values built into SYSBOOT. SYSBOOT calculates other system parameters whose values depend on the values of the adjustable parameters.

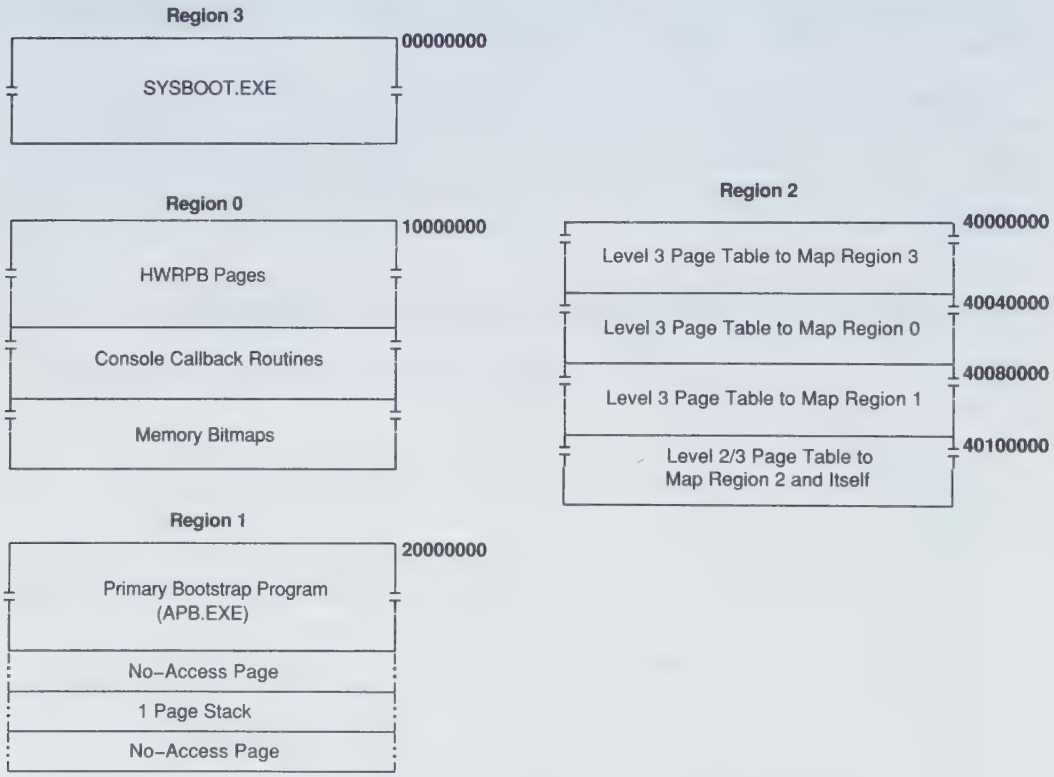
2. SYSBOOT sets up the SPT to map system virtual address space. The sizes of many pieces of system address space depend on the values of one or more SYSGEN parameters. The calculations that SYSBOOT performs and the results of these calculations are detailed in Appendix *Size of System and P1 Virtual Address Spaces*.

3. After it has established system virtual address space, SYSBOOT calls LDR$LOAD_IMAGE, in module SYSLDR, to load the following images into memory:
   - — The system base images SYS$PUBLIC_VECTORS.EXE and SYS.EXE
   - — The system-specific image SYS$CPU_ROUTINES_*xxyy*, where *xx* is the system type designator and *yy* is the processor type designator

     Note that the image SYS$CPU_ROUTINES_*xxyy* is functionally equivalent to the image SYSLOA*xxx* in VAX/VMS.
   - — The executive images that contain the boot drivers for the system device
   - — A number of executive images common to all systems (see Table 33–1)

SYSBOOT is initiated and runs in kernel mode at IPL 31 in bootstrap space in the boot HWPCB's context. APB loads SYSBOOT into region 3 of bootstrap space, which starts at virtual address 0. Figure 33–14 shows bootstrap virtual address space after SYSBOOT is loaded.

Figure 33–14 Bootstrap Virtual Address Space after SYSBOOT Is Loaded



SYSBOOT is built from a number of modules in facilities [SYSBOOT], [APB], and [SYS]. SYSBOOT has its own copies of APB's CPU-specific routines, primitive file system routines, primitive exception handling code, and memory allocation routines. SYSBOOT is linked with XDELTA.

Note that SYSBOOT uses APB's boot drivers to load executive images.

33.4.1 SYSBOOT Flow

APB calls BOO$EVMS_SYSBOOT in module [SYSBOOT]SYSBOOT, passing it a single argument, the address of the argument list described in Table 33–10. BOO$EVMS_SYSBOOT is SYSBOOT's main procedure. It performs the following steps:

1. It sets BOOSTATE$V_SYSBOOT in the location EXE$GL_STATE to indicate that secondary bootstrap is in progress.
2. It copies a number of cells from the argument list to locations within the SYSBOOT image that are referenced by SYSBOOT's own memory management routines.
3. APB operates in region 0 of bootstrap virtual address space, whereas SYSBOOT operates in region 3. However, SYSBOOT uses a number of APB routines. To establish an environment for these routines in region 3 of bootstrap virtual address space, SYSBOOT repeats steps 1, 2, 3, 5, 10, and 11 in Section 33.3.1.
4. It sets SWRPB_BOOT_FLAGS$V_NOHEADER in the SWRPB to indicate that LDR$LOAD_IMAGE must read executive image headers.
5. It loads the SYSGEN parameter file, VAXVMSSYS.PAR. Chapter *Operating System Initialization and Shutdown* describes in detail the movement of parameter information during the initialization sequence.
6. If a conversational bootstrap was requested (the CONV boot flag is set), SYSBOOT calls BOO$GETPARAM, in module [SYSBOOT]SYSBOOCMD. BOO$GETPARAM allows the console operator interactively to examine and modify SYSGEN parameters.
7. Many [*censored*] SYSGEN parameters are specified in units of VAX pages. A VAX page is 512 bytes in size. The size of a [*censored*] page is processor-dependent and can be 8K, 16K, 32K, or 64K bytes in size. On all current [*censored*] processors the page size is 8K bytes. SYSBOOT converts parameters that are specified in units of VAX pages, or pagelets, into units of [*censored*] pages.
8. It allocates a page of physical memory and records its PFN in the last PTE of the L1PT for bootstrap virtual address space. It then makes that PTE valid. This makes the newly allocated page the level 2 page table (L2PT) of system virtual address space.

   At this point, system virtual address space comes into existence. Until the PTEs in the L2PT are filled in later by SYSBOOT, however, all pages in system virtual address space are invalid.
9. SYSBOOT calls LDR$INIT_MEM, in module LDR_MEM_INIT. If the SYSGEN parameter LOADABLE_SYS indicates that executive image slicing is enabled, LDR$INIT_MEM reserves two physically and virtually contiguous regions of address space (known as huge pages) for later loading of executive images. Regardless of the value of LOADABLE_SYS, LDR$INIT_MEM reserves one huge page for nonpaged system data such as the PFN database.

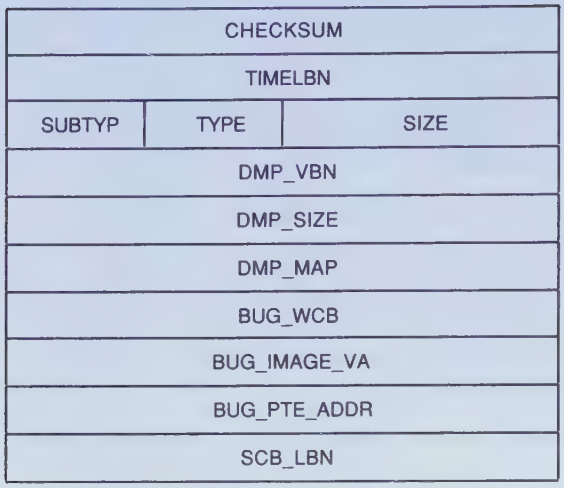
Use of such regions improves performance by reducing translation buffer misses. Chapter *The Modular Executive* contains more information.

10. It loads the base system images SYS$PUBLIC_VECTORS.EXE and SYS.EXE into memory and initializes the following cells:
    - EXE$GL_SYS_SYMVEC—Base of SYS.EXE's symbol vector
    - EXE$GL_SYS_SYMVEC_END—End of SYS.EXE's symbol vector
    - EXE$GL_SYS_CODE_BASE—Base of SYS.EXE's code area
    - EXE$GL_SYS_CODE_END—End of SYS.EXE's code area
    - EXE$GL_SYS_DATA_BASE—Base of SYS.EXE's data area
    - EXE$GL_SYS_DATA_END—End of SYS.EXE's data area
    - EXE$GL_PUBLIC_VECTOR_SYMVEC—Base of SYS$PUBLIC_ VECTORS's symbol vector
    - EXE$GL_PUBLIC_VECTOR_SYMVEC_END—End of SYS$PUBLIC_ VECTORS's symbol vector
11. Once the base system images have been loaded, SYSBOOT fixes up its own references to locations in them. To do so, it calls UPDATE_SHL_SYSBOOT, in module SYSLDR_COMMON, and FIXUP_IMAGE, in module RELOCATE_ AND_FIXUP.

    After the SYSBOOT image is fixed up, its image header is no longer required; SYSBOOT deallocates it.
12. It initializes miscellaneous data cells used for memory management and executive image loading.
13. It sizes and allocates system virtual memory for the balance set slots. No physical memory is allocated. Appendix *Size of System and P1 Virtual Address Spaces* contains more details on the sizing of balance set slots.
14. SYSBOOT allocates system virtual memory for the global page table (GPT). No physical memory is allocated. Chapter *Memory Management Data Structures* provides more information on the GPT.
15. It allocates and initializes a piece of the nonpaged system data huge page, called a slice (see Chapter *The Modular Executive*), for the system header. This step is analogous to the process header configuration performed by code in the shell as a part of process creation (see Chapter *Process Creation*).
16. It sizes, allocates, and initializes a slice of the nonpaged system data huge page for the PFN database. Appendix *Size of System and P1 Virtual Address Spaces* contains details of the calculations.
17. It allocates and initializes a slice of the nonpaged system data huge page for the nonpaged pool allocation region. If the SYSGEN parameter POOLPAGING is set to zero, it allocates and initializes virtual and physical memory for the paged pool allocation region also. Otherwise, it simply allocates address space for the paged pool allocation region.

18. SYSBOOT allocates 8K bytes of virtual and physical memory for the SCB, records the SCB starting virtual address in EXE$GL_SCB, and copies the SCB template at location SCB$A_BASE, in module SCBVECTOR, to the newly allocated SCB.
19. It allocates the processor's per-CPU database from nonpaged pool, clears it, and records its address in the processor base register (PRBR). Section 33.4.2 describes the per-CPU database in detail.
20. It copies the primitive file cache created by APB into nonpaged pool.
21. SYSBOOT remaps the HWRPB from bootstrap space into system space and records its address in EXE$GPQ_HWRPB. It copies the SWRPB from bootstrap space into nonpaged pool and records its address in EXE$GPQ_SWRPB. It remaps console callback routines into system space and records their new addresses in the HWRPB_CRB.
22. SYSBOOT remaps the tested physical memory bitmaps from bootstrap space into system space.
23. It initializes the per-CPU database by performing the following steps:
    a. It saves the virtual address of the per-CPU slot in the per-CPU database.
    b. It allocates three pages of virtual memory and one page of physical memory for a one-page kernel stack and its two guard pages, one on each side. It records the low-end address of the kernel stack in the kernel stack pointer register in the system HWPCB in the per-CPU database (see Section 33.4.2.2). It also initializes the executive, supervisor, and user mode stack pointer registers in the system HWPCB to point to one of the guard pages.
    c. It records the PFN of the L1PT in the system HWPCB's HWPCB$IQ_PTBR. It maps the L1PT into system space and records the L1PT's virtual address in MMG$GQ_PT_VA and in HWPCB$IQ_PT_VA.
24. It allocates the boot control block (BOOTCB), shown in Figure 33–15, from nonpaged pool and initializes it. It records the address of the BOOTCB in EXE$GL_BOOTCB.
    a. If the system device is a disk, SYSBOOT opens and maps the system dump file, [SYS*n*.SYSEXE]SYSDUMP.DMP, and records the file's mapping information for use in the event of a bugcheck.
    b. If SYSBOOT does not find the dump file, it opens and maps the primary page file, [SYS*n*.SYSEXE]PAGEFILE.SYS, and sets the flag EXE$V_PAGFILDMP in EXE$GL_DEFFLAGS. The first blocks of the page file, if one exists, are used as an alternative dump file when the system bugchecks. When the SYSINIT process runs (see Chapter *Operating System Initialization and Shutdown*), it will look in the page file instead of the dump file for saved error log messages to restore.
    c. SYSBOOT records the LBN of the block in SYS.EXE that contains the cell EXE$GQ_TODCBASE.

Figure 33–15 Boot Control Block



25. SYSBOOT loads one or more executive images for boot drivers selected by APB. The boot drivers built into APB.EXE operate in bootstrap space and are used only by APB and SYSBOOT. The ones loaded in this step operate in system space and are used by EXE$INIT and VMS bugcheck processing code.
26. It selects executive images to be loaded in the next step and adds a file statistics block pointer to the array BOO$IMAGE_BLOCK for each one. Table 33–1 lists the executive images that SYSBOOT loads.
27. SYSBOOT loops through the array BOO$IMAGE_BLOCK, calling LDR$LOAD_IMAGE to load each executive image selected in the step 26.
28. It initializes the boot parameter argument list (BOOPAR), shown in Figure 33–16. This is the argument list that SYSBOOT passes to EXE$INIT.
29. It calls the boot driver's unit disconnect routine because there is no longer any use for the boot driver that is linked into APB. EXE$INIT will perform the initialization necessary for the system device's boot driver executive image.
30. If the BOOBPT boot flag is set, SYSBOOT executes a `CALL_PAL BPT` instruction at a location known to XDELTA. As a result of the exception, control transfers to XDELTA.
31. If SYSTEM_DEBUG.EXE has been loaded, SYSBOOT calls XDT$SYSDBG_INIT, in module [DELTA]XDELTA. This informs XDELTA that INI$BRK, in module SYSTEM_ROUTINES, and not BOO$BRK, is to be the permanent XDELTA breakpoint.
32. It executes the `CALL_PAL SWPCTX` instruction to swap from the bootstrap HWPCB to the system HWPCB. At this point, virtual address space is laid out as shown in Figure 33–17.

Figure 33–16 Boot Parameter Argument List

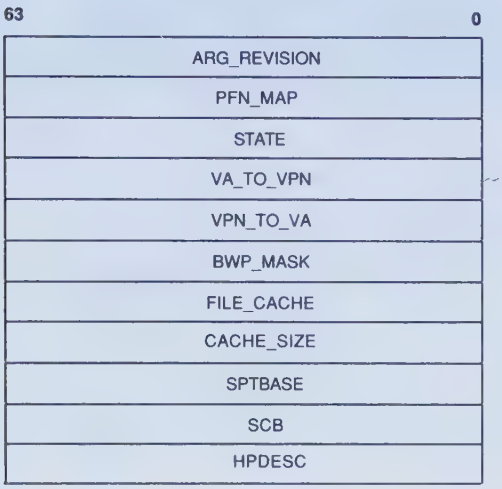
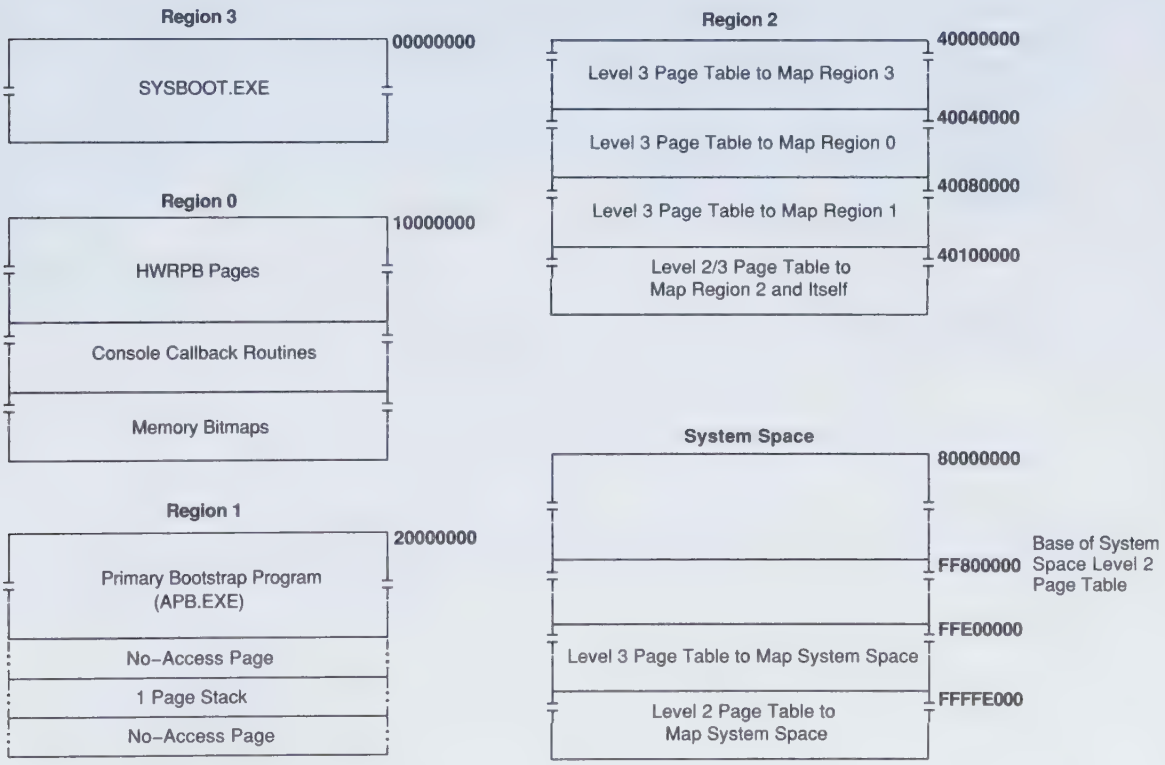


Figure 33–17 Virtual Address Space at the End of SYSBOOT



33. As its final step, SYSBOOT calls EXE$INIT, in module INIT, to begin initialization of the executive. Note that early in the execution of EXE$INIT, the physical memory occupied by regions 0 through 3 are freed up and the virtual address space occupied by these regions is deleted; only system space remains.

33.4.2 CPU-Specific Data

Each member of a VAX/VMS SMP system has memory for data that describes the state of that CPU. Although [*censored*] VMS Version 1.0 does not support SMP, CPU-specific data is maintained within both a per-CPU slot in the HWRPB and a nonpaged pool data structure called the per-CPU database. The HWRPB, described in Section 33.2.5, contains CPU-specific data that is architecturally defined. The per-CPU database contains software-defined data specific to one CPU. Leaving software-defined CPU-specific data in one data structure simplified the porting of VMS from the VAX to the [*censored*] architecture and simplifies the possible future addition of SMP support.

33.4.2.1 Locating Per-CPU Data

The [*censored*] architecture defines the processor base register to contain CPU-specific data. VMS loads the virtual address of the per-CPU database into the PRBR. Code running in kernel mode can read the register by executing the CALL_PAL MFPR_PRBR instruction or by invoking a language-specific FIND_CPU_DATA macro.

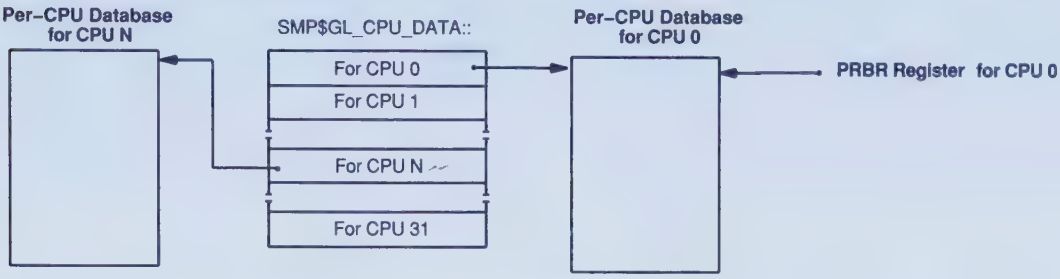
An example of a VAX MACRO FIND_CPU_DATA macro invocation and its expansion follows:

```
; Macro invocation
;
        FIND_CPU_DATA R4            ;Get per-CPU database address in R4
;
; Macro expansion
;
        $PRDEF                      ;Define processor register numbers
        MFPR      #PR$_PRBR,R4
        .SET_REGISTERS ALIGNED=R4 ;Tell MACRO that R4
                                  ; contents are aligned
```

Note that use of this macro or instruction is restricted to code that executes in kernel mode. Furthermore, the code must run at an IPL above 2 between getting and using the address of the per-CPU database. If SMP were supported, code running at IPL 2 or below would be subject to rescheduling and subsequent execution on another processor whose per-CPU data area is at a different address.

In an SMP configuration, an array called the CPU data vector contains the addresses of the per-CPU databases of the individual members. The 32-longword array begins at global SMP$GL_CPU_DATA. It is indexed by CPU ID number to get the address of the database for a particular processor. Figure 33–18 shows the relation between the per-CPU database and the CPU data vector.

Figure 33–18 CPU Data Vector and Per-CPU Database



33.4.2.2 Per-CPU Stack

The [*censored*] architecture defines a per-process stack for each of the four access modes in which the CPU can execute but, unlike the VAX architecture, does not include an interrupt stack. Executing in process context, a processor runs on an access mode stack private to that process. Executing in the context of a system or termination HWPCB, a processor executes on the kernel stack specified by that HWPCB. On an SMP system, multiple processors could each execute in its own system or termination context at the same time.

Simultaneous use of a stack or HWPCB by more than one processor is not viable. The console subsystem therefore provides for the possibility of a boot HWPCB for each processor in an area of the HWRPB called a per-CPU slot; [*censored*] VMS provides for the possibility of a system HWPCB and termination HWPCB for exclusive use by a processor. Figure 33–19 shows the relations among the per-CPU database, the per-CPU slot, and the per-CPU HWPCBs and their associated stacks.

33.4.2.3 Per-CPU Database

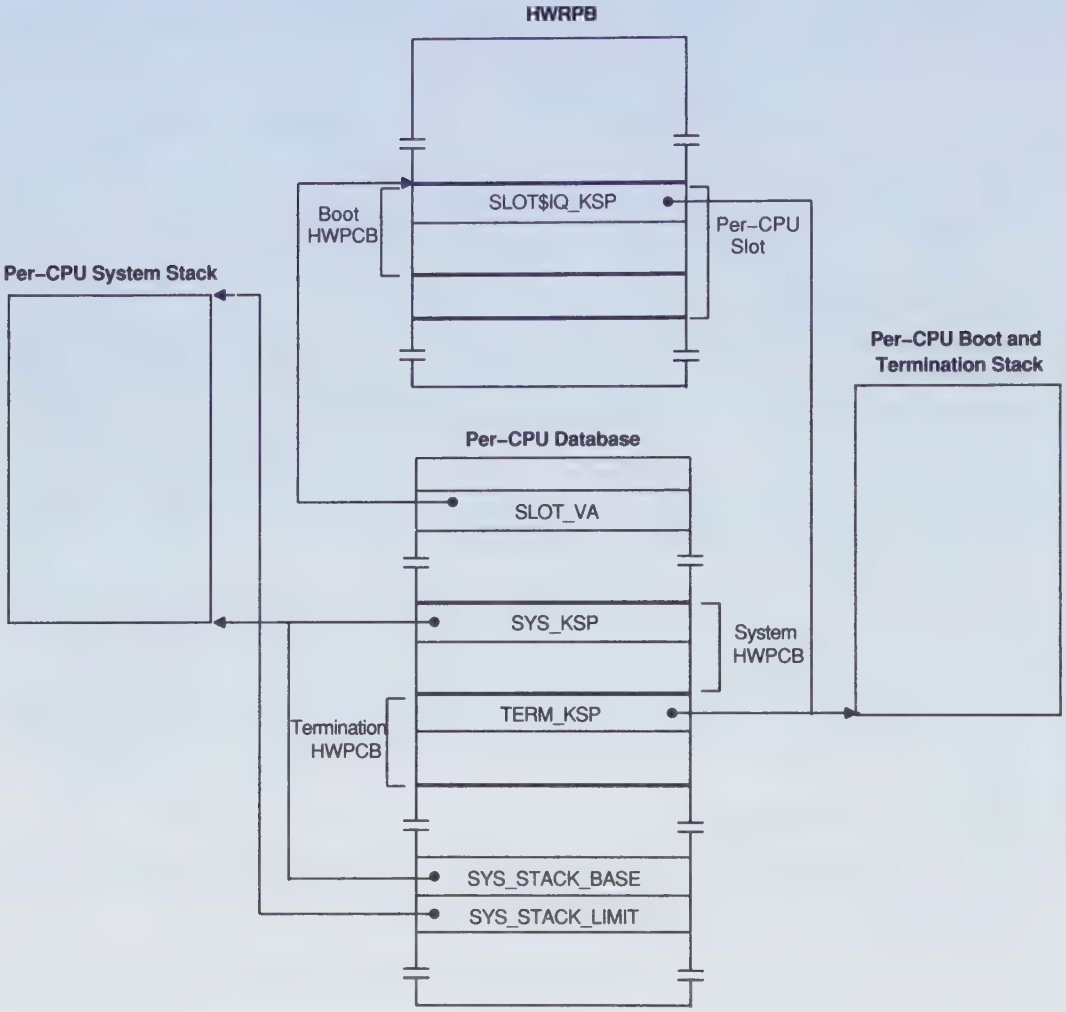
The per-CPU database contains processor-specific information such as the process control block (PCB) address and priority of its current process, its CPU ID, and its fork queues. Figure 33–20 shows the per-CPU database.

The per-CPU database can be extended to include data specific to a particular processor type. By default there is no extension. If the structure were extended, CPU$L_VARIABLE_OFFSET would contain the offset from the beginning of the structure to the CPU-type-specific data, and CPU$L_VARIABLE_LENGTH would contain the size in bytes of the extension. Currently, no CPU requires use of this extension, and these two fields contain zero.

CPU$L_CURPCB contains the PCB address of the process currently executing on this processor. CPU$B_CUR_PRI contains the process's current priority. (This field has been expanded to a longword and is also called CPU$L_CUR_PRI.) When the CPU is idle, CPU$L_CURPCB contains the address of the null PCB, and CPU$B_CUR_PRI contains –1.

CPU$L_SLOT_VA contains the virtual address of the per-CPU slot in the HWRPB.

Figure 33–19 Per-CPU Contexts and Stacks



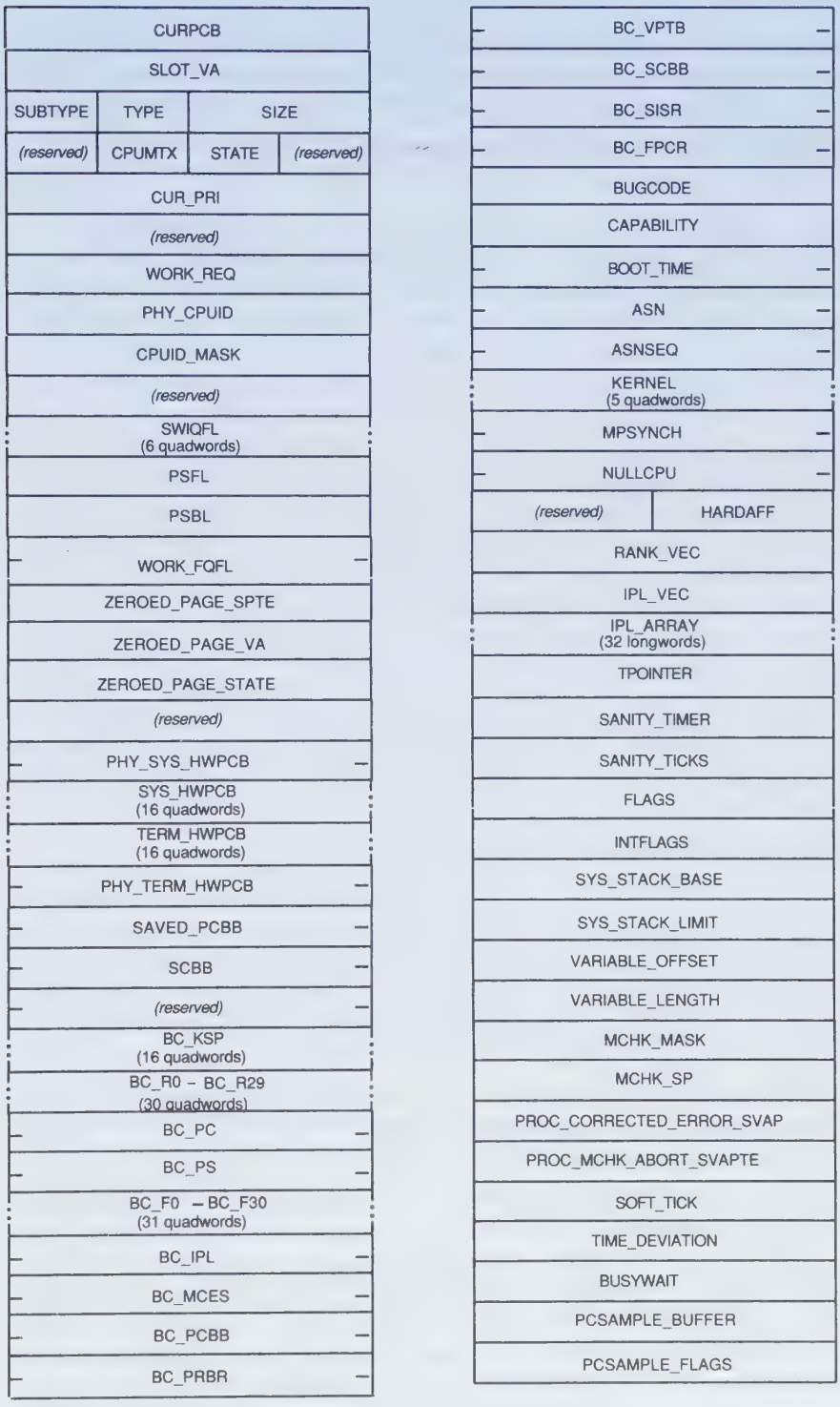
CPUW_SIZE, CPUB_TYPE, and CPU$B_SUBTYPE contain the standard dynamic data structure header fields. CPU$W_SIZE contains the length in bytes of only the standard part of the structure, which does not include the length of a processor-type-specific extension.

CPU$B_STATE identifies the processor's current state, for example, CPU$C_ BOOTED.

CPU$B_CPUMTX is the number of times the CPU mutex has been acquired by the processor; the CPU mutex can be acquired multiple times by nested routines.

CPU$L_WORK_REQ is a bit mask describing outstanding work requests made by other processors of this processor.

Figure 33–20 Per-CPU Database



(continued)

CPU$L_PHY_CPUID contains the ID of the processor, a number from 0 to 31. CPU$L_CPUID_MASK is a mask of 31 zeros with a single bit set in the bit position corresponding to the CPU ID number.

The processor's fork dispatching queues begin at CPU$Q_SWIQFL. Chapter *Software Interrupts* describes fork dispatching and the use of fork queues. The per-processor I/O postprocessing queue is at CPU$L_PSFL and CPU$L_PSBL.

CPU$Q_WORK_FQFL is a work queue for switching fork processes from other processors to this one.

During system initialization, the scheduling initialization routine allocates an SPTE and stores its address in CPU$L_ZEROED_PAGE_SPTE. It stores the system virtual address the SPTE maps in CPU$L_ZEROED_PAGE_VA. During normal system operation, when there is no computable process to execute, the scheduler idle loop routine clears unused pages of memory for future allocation. Clearing them in the idle loop minimizes the time required for a process to allocate a demand zero page.

The idle loop code maps a free physical page to be cleared with the SPTE in CPU$L_ZEROED_PAGE_SPTE and accesses it through the virtual address in CPU$L_ZEROED_PAGE_VA. Whenever a process becomes computable, the idle loop code discontinues zeroing and records how many 32-byte chunks in the page have not yet been zeroed in CPU$L_ZEROED_PAGE_STATE. On an SMP system, multiple processors could execute the idle loop and thus each would need its own SPTE and place to record its progress. Chapter *Paging Dynamics* describes this mechanism in more detail.

The per-CPU database contains two HWPCBs: the sytem HWPCB begins at field CPU$Q_SYS_HWPCB, and the termination HWPCB at field CPU$Q_TERM_HWPCB. CPU$Q_PHY_SYS_HWPCB and CPU$Q_PHY_TERM_HWPCB contain their physical addresses.

CPU$Q_SCBB contains the physical address of the SCB.

During a powerfail, the current contents of various volatile registers are stored in the per-CPU database so that they can be restored during restart. During fatal bugcheck processing, the current contents of various volatile registers are stored in the per-CPU database so that they can be examined during crash dump analysis. Chapters *Power Failure and Recovery* and *Error Handling* describe powerfail and bugcheck processing and use of these fields.

Generally, powerfail and bugcheck processing save the same set of registers in the same per-CPU database fields. These per-CPU database fields and their contents are as follows:

- CPU$Q_SAVED_PCBB—The physical address of the current process's hardware PCB at the time of a powerfail
- CPU$Q_BC_KSP and the next 15 quadwords—The contents of the registers whose save area is in the HWPCB at the time of a bugcheck
- CPU$Q_BC_R*x*, where *x* is a number from 0 to 29—The contents of the integer registers at the time of a powerfail or bugcheck
- CPU$Q_BC_PC and CPU$Q_BC_PS—The contents of the PC and PS at the time of a powerfail or bugcheck

- CPU$Q_BC_F*x*, where *x* is a number from 0 to 30—The contents of the floating-point registers at the time of a powerfail or bugcheck (if the current process is using floating-point)
- CPU$Q_BC_PCBB—The physical address of the current process's hardware PCB at the time of a bugcheck
- CPUQ_BC_IPL, CPUQ_BC_MCES, CPU$Q_BC_SCBB, and CPU$Q_BC_VPTB—The contents of these processor registers (IPL, machine check error summary, system control block base, and virtual page table base) at the time of a bugcheck
- CPUQ_BC_SISR, CPUQ_BC_FPCR, and CPU$Q_BC_PRBR—The contents of these processor registers (software interrupt summary register, floating-point control register, and processor base register) at the time of powerfail or a bugcheck

CPU$L_BUGCODE contains the bugcheck code for a CPU that is crashing.

CPU$L_CAPABILITY is a bit mask with bits set to represent the capabilities of this processor. The low bit, when set, means that this CPU is the primary processor. The macro $CPBDEF defines symbolic values for the bits in this field. CPU$W_HARDAFF is the number of processes that have explicit affinity for this CPU. Chapter *Scheduling* describes processor capabilities and process affinity.

CPU$Q_BOOT_TIME contains the system time at which the CPU was booted.

When process context is switched, per-process virtual address translations that are cached in the translation buffer must be flushed to preclude stale translations in the context of the next process. To enable per-process address translations to remain cached, the [*censored*] architecture supports the association of a process address space number with each per-process translation. CPU$Q_ASN and CPU$Q_ASNSEQ describe the address space number associated with the current process or, if there is none, the most recent one. Chapter *Memory Management Data Structures* describes translation buffer invalidation and address space numbers in more detail.

Beginning at field CPU$Q_KERNEL is a five-quadword array that records the amount of time the processor executes in each access mode; the first four elements are for process context time in each mode, and the fifth is for time in system context. CPU$Q_KERNEL does not contain time spent in a spinwait: CPU$Q_MPSYNCH records the amount of time the processor spins waiting to acquire a spinlock from a process-context kernel mode thread of execution. CPU$Q_NULLCPU records the amount of time spent in the scheduler idle loop. These counts are maintained by the interval timer interrupt service routine, described in Chapter *Time Support.*

Each bit, excluding bit 31, set in the field CPU$L_RANK_VEC corresponds to a static spinlock held by the processor; its bit position identifies the spinlock rank. Each bit set in the field CPU$L_IPL_VEC corresponds to an IPL at which the processor holds one or more spinlocks. The IPL representation is inverted. When a processor acquires a spinlock, the IPL of the spinlock is subtracted from 31. The bit in CPU$L_IPL_VEC corresponding to that number is set. The field thus represents the current set of (inverted) spinlock IPLs active on the processor. The inverted number is also used as an index into the 32-longword array at CPU$L_

IPL_ARRAY, which records the number of different spinlocks held at each IPL. These fields are provided to facilitate support for a full-checking version of the spinlock routines, in case of the possible future addition of SMP support.

The fields CPU$L_TPOINTER, CPU$L_SANITY_TIMER, and CPU$L_SANITY_TICKS are part of the SMP sanity timeout mechanism.

CPU$L_FLAGS and CPU$L_INTFLAGS contain various flags. One such flag is CPU$V_STOPPING, which, when set, means that the processor is stopping. The flags in CPU$L_INTFLAGS are set and cleared with VAX MACRO interlocked instructions.

CPU$L_SYS_STACK_BASE and CPU$L_SYS_STACK_LIMIT contain the high and low addresses of the kernel stack on which the processor runs during normal operations when there is no current process to execute. It is the stack identified by the system HWPCB.

CPU$L_MCHK_MASK and CPU$L_MCHK_SP are related to the machine check protection mechanism, which inhibits VMS's default reaction to a nonexistent memory machine check. Autoconfiguration code enables this mechanism while probing I/O space to determine which devices are present.

During system initialization, a CPU-specific initialization routine allocates two SPTEs and stores their addresses in CPU$L_PROC_CORRECTED_ERROR_SVAP and CPU$L_PROC_MCHK_ABORT_SVAPTE. When either a corrected error or machine check abort occurs, information about the error is collected in an area of physical memory called a logout area. The error service routine is entered with the address of the logout area. To access the logout area, the error service routine maps it with one of the reserved SPTEs: service routines for corrected error interrupts use the one identified by CPU$L_PROC_CORRECTED_ERROR_SVAP; service routines for machine check aborts use the one identified by CPU$L_PROC_MCHK_ABORT_SVAPTE.

CPU$L_SOFT_TICK and CPU$L_TIME_DEVIATION are part of the implementation of "soft" 10-millisecond ticks. Because the interval timer interrupts at a CPU-specific frequency that is not necessarily an integral factor of 10 milliseconds, [*censored*] VMS must simulate a 10-millisecond interval to provide compatibility with VAX/VMS. In CPU$L_SOFT_TICK, VMS maintains the number of timer interrupts left until the current calculated 10-millisecond interval is over. When the interval is over, VMS checks how close the approximation is to true time. CPU$L_TIME_DEVIATION is a running count used to determine whether an additional tick is necessary to average out the length of the simulated intervals. Chapter *Time Support* contains further information.

CPU$L_BUSYWAIT is nonzero while the processor is spinning, trying to acquire a spinlock, as described in Chapter *Synchronization Techniques*. While this field is nonzero, the interval timer interrupt service routine does not charge a timer tick against the quantum of the current process (see Chapter *Time Support*). It also uses this field to determine whether to record a process context kernel mode tick in CPU$Q_KERNEL or CPU$Q_MPSYNCH.

CPU$L_PCSAMPLE_BUFFER and CPU$L_PCSAMPLE_FLAGS support PC sampling, a mechanism for monitoring performance. CPU$L_PCSAMPLE_BUFFER, when nonzero, contains the address of a nonpaged pool data structure in which regularly sampled PC values are stored. A monitoring process periodically collects the PC samples. A bit in CPU$L_PCSAMPLE_FLAGS indicates whether a PC sample is being stored. The bit is cleared to indicate that the data structure is in a stable state from which PC samples can be retrieved.

33.5 Revision Information

Source Module	Build Date
[APB]BDTAB.SDL	23-Oct-1991
[APB]BDTDEF.SDL	23-Oct-1991
[APB]BDVDEF.SDL	23-Oct-1991
[APB]BOODEF.SDL	23-Oct-1991
[APB]BOOT_FLAGS.BLI	23-Oct-1991
[APB]BTDRIVER_INIT.BLI	23-Oct-1991
[APB]BTDRIVER_SUPPORT.BLI	23-Oct-1991
[APB]CPU_VECTOR.BLI	23-Oct-1991
[APB]DKBTDRIVER.BLI	23-Oct-1991
[APB]FILE_SYSTEM.BLI	23-Oct-1991
[APB]HWRPBDEF.SDL	23-Oct-1991
[APB]MAIN.BLI	23-Oct-1991
[SYSBOOT]SYSBOOT.BLI	14-Dec-1991
[SYSBOOT]SYSBOOT64.B64	14-Dec-1991
[LIB]CPUDEF.SDL	31-Jan-1992

Appendixes

Appendix D
Naming Conventions

The conventions described in this appendix were adopted to aid implementors in producing meaningful public names. Public names are names that are global (known to the linker) or that appear in parameter or macro definition files. Public names follow these conventions for the following reasons:

- Using reserved names ensures that customer-written software will not be invalidated by subsequent releases of Digital products that add new symbols.
- Using definite patterns for different uses tells someone reading the source code what type of object is being referenced. For example, the form of a macro name is different from that of an offset, which is different from that of a status code.
- Using length codes within a pattern associates the size of an object with its name, increasing the likelihood that reference to this object will use the correct VAX MACRO instructions.

  As the use of VAX MACRO decreases, it is expected that the use of length codes will decline. Software writers are encouraged not to explicitly specify the data size of a variable unless the situation demands it (see Section D.1).
- Using a facility code in symbol definitions gives the reader an indication of where the symbol is defined. Separate groups of implementors choose facility code names that will not conflict with one another.

To fully conform with these standards, local synonyms should never be defined for public symbols. The full public symbol should be used in every reference to give maximum clarity to the reader.

D.1 Public Symbol Patterns

All Digital symbols contain a dollar sign. Thus, customers and applications developers are strongly advised to use underscores instead of dollar signs to avoid potential conflicts.

Public symbols should be constructed to convey as much information as possible about the entities they name. Frequently, private names follow a similar convention. The private name convention is then the same as the public one, with the underscore replacing the dollar sign in symbol names. Private names are used both within a module and globally between modules of a facility that is never in a library. All names that might ever be bound into a user's program must follow

the rules for public names. In the case of internal names, a double dollar sign convention can be used, as shown in item 4 in the following list of formats:

1. System service and Record Management Services (RMS) VAX MACRO names are of the form

   $service-name

   In a system service VAX MACRO name, a trailing _S or _G distinguishes the stack form from the separate argument list form. Details about the names of system service macros can be found in the *Introduction to VMS System Services*.

   These names appear in the system macro library STARLET.MLB and represent a call to one of the VMS system services or RMS services. The following examples show this form of symbol name:

$ASCEFC_S	Associate common event flag cluster
$CLOSE	Close a file
$TRNLNM_G	Translate logical name

2. Facility-specific public macro names are of the form

   $facility_macro-name

   The executive does not use any symbol names of this form.

3. System macros using local symbols or macros always use names of the form

   $facility$macro-name

   This is the form to be used both for symbols generated by a macro and included in calls to it, and for internal macros that are not documented. The executive does not use any symbol names of this form.

4. Global entry point names are of the form

   facility$entry-name

   The following examples show this form of symbol name:

EXE$ALOPAGED	Allocate paged pool
IOC$KP_WFIKPCH	Wait for interrupt and keep channel
MMG$PAGEFAULT	Page fault handler

   Global entry point names that are intended for use only within a set of related procedures but not by any calling programs outside the set are of the form

   facility$$entry-name

   The executive contains few symbol names of this form. However, the Run-Time Library contains several examples of symbol names that follow this convention, for example:

BAS$$STOP	Signal a BASIC fatal error
FOR$$SIGNAL_STO	Signal a FORTRAN error and call LIB$STOP
OTS$$CLOSE_FILE	Internal routine to close file

5. Global entry point names that have nonstandard invocations (VAX MACRO JSB entry point names) are of the following form, where _R*n* indicates that R0 through R*n* are not preserved by the routine:

   facility$*entry-name*_R*n*

   In VAX/VMS, the invoker of such an entry point must include at least registers R2 through R*n* in its own entry mask so that all registers used can be restored properly. For [*censored*] VMS, these routines are no longer supported. Users of a Run-Time Library (RTL) routine of this form must call the corresponding routine whose name is derived by dropping the suffix _R*n*. Until code changes are actually made, the translated form of these routines in the VAX environment software translator (VEST) RTL can be used. The VEST RTL contains several examples of such routines:

LIB$ANALYZE_SDESC_R2	Analyze string descriptor
OTS$MOVE3_R5	Copy string without any fill characters
STR$COPY_DX_R8	JSB entry to general string-copying routine

6. Status codes and condition values are of the form

   facility$_*status*

   The following examples show this form of symbol name:

RMS$_FNF	File not found
SS$_ILLEFC	Illegal event flag cluster
SS$_WASCLR	Flag was previously clear

7. Global variable names are of the form

   facility$G*t*_*variable-name*

   The letter G indicates a global variable. The letter *t* represents the type of variable (see Section D.2). The following examples show this form of symbol name:

CTL$GQ_PROCPRIV	Process privilege mask
EXE$GQ_SYSTIME	System date and time
SCH$GL_FREECNT	Number of pages on the free page list

8. Addressable global arrays use the letter A (instead of the letter G) and are of the form

   facility$A*t*_*array-name*

   The letter A indicates a global array. The letter *t* indicates the type of array element (see Section D.2). In some uses, the symbol's value is the address of the beginning of the array; in other uses, the symbol is the name of a variable that contains the address of the beginning of the array. The following examples show both uses of this form of symbol name:

CTL$AQ_EXCVEC	Array of primary and secondary exception vectors
CTL$AL_STACK	Array of stack limits
EXE$AL_ERLBUFADR	Address of array of error log allocation buffers

9. The letter A, along with the letter R, indicates a vectored universal symbol in the SYS.EXE base image that contains the address of an executive image data structure. Chapter *The Modular Executive* describes the modular organization of the VMS executive in detail. The following examples show this form of symbol name:

EXE$AR_NPOOL_DATA	Address of data related to nonpaged pool allocation
SYS$AR_JOBCTLMB	Address of job controller's mailbox unit control block

10. New for [*censored*] VMS, variables of the following form are pointers:

    structure$P*x*_*variable-name*

    where *x* stands for S, H, L, or Q. For most variables, *x* stands for S or H, the software-defined or hardware-defined default pointer size. In such a form, the software writer makes no assumption about pointer size. This is the preferred form of pointer definition for new data structures.

    Where the size of a pointer must be explicitly specified as a longword or quadword, *x* stands for L or Q. This form of variable has been useful for porting VAX/VMS.

    To facilitate development of [*censored*] VMS, many pointer variables have been allowed to remain as they are rather than being changed to conform to this convention.

    The following are examples of the form *structure*$P*x*_*variable-name*:

ADP$PS_MBPR	Address of mailbox pointer register (in adapter control block)
ICB$PH_PROGRAM_COUNTER	Address of the program counter (in the invocation context block of the translated image environment)
SYSG$PQ_CTB_PTR	Address of console terminal block (in the SYSGEN utility data block)
SYSG$PL_CTB_PTR_L	Low-order longword of the console terminal block address

    Note that SYSG$PQ_CTB_PTR and SYSG$PL_CTB_PTR_L address the same offset in the SYSGEN data block, although the latter addresses only the low-order longword. This convention of suffixing a variable name with an _L to address the low-order longword part of a larger structure is widely used in [*censored*] VMS. Similarly, the suffix _H is used to address the high-order longword.

11. New for [*censored*] VMS, variables of the following form are integer variables:

    structure$I*x*_*variable-name*

    where *x* is the same as *t* for standard-sized variables (B, W, L, or Q) and *x* stands for S or H for integers whose size is the software-defined or hardware-defined default. Unless an integer variable must be of a specific size, it is defined using the S or H form.

Public structure offset names new for [*censored*] VMS are also defined in this form. The following are examples of variables of this form:

KPB$IS_STACK_SIZE	Stack size (in the kernel process block)
CHF$IH_MCH_SAVR0	Register R0 save area (in the mechanism array)
HWPCB$IQ_KSP	Kernel stack pointer (in the hardware privileged context block)
DDB$IL_ALLOCLS	Device allocation class (in the device data block)
DPT$IW_INAME_LEN	Length of driver's image name (in the driver prologue table)
KPB$IB_TYPE	Data structure type (in the kernel process block)

Most offset names that have been modified to adhere to the new convention have also retained their old names as synonyms for ease of porting VAX/VMS. For example, DDB$L_ALLOCLS is a synonym for DDB$IL_ALLOCLS.

12. Also new for [*censored*] VMS, floating-point variables and public structure offset names for floating-point data are of the following form:

    structure$FH_*variable-name*

    The following is an example of this form:

CHF$FH_MCH_SAVF29	Save area for floating-point register F29 (in mechanism array)

13. Public structure definition macro names are of the form

    $*facility_structure*DEF

    Invoking this macro defines all symbols of the form *structure*$*xxxxxx*.

    Most of the public structure definitions used by the VMS operating system do not include the string *facility_* in the macros that define structure offsets. Rather, macros of the following form are used to define *structure*$*xxxxxx* symbols:

    $*structure*DEF

    The following examples show the $*structure*DEF form of the macro:

$ACBDEF	Offsets into asynchronous system trap (AST) control block
$PCBDEF	Offsets into software process control block
$PHDDEF	Offsets into process header

    Many of the macros of this form are contained in the macro libraries LIB.MLB or STARLET.MLB. These macros are initially defined in a language-independent structure definition language, as described in Appendix *Use of Listings and Map Files*.

14. Public structure offset names are of the form

    structure$*t_field-name*

The letter *t* indicates the data type of the field (see Section D.2). The value of the public symbol is the byte offset to the start of the data element in the structure. The following examples show this form of symbol name:

CEB$L_EFC	Event flag cluster (in common event block)
GSD$W_SIZE	Size of global section descriptor in bytes
PCB$B_PGFLINDEX	Process page file index (in software process control block)

15. VAX MACRO public structure bit field offsets and single bit names are of the form

    structure$V_*field-name*

    The value of the public symbol is the bit offset from the start of the field that contains the data, not from the start of the control block. The following examples show this form of symbol name:

ACB$V_QUOTA	Charge AST to process AST quota
CRB$V_BSY	Controller is busy (in channel request block)
UCB$V_CANCEL	Cancel I/O on this unit

16. VAX MACRO public structure bit field size names are of the form

    structure$S_*field-name*

    The value of the public symbol is the number of bits in the field. The following examples show this form of symbol name:

ACB$S_MODE	Length of requestor's access mode field (2 bits)
VA$S_BYTES_PER_PAGELET	Length of bytes per pagelet field (9 bits)
PTE$S_PROT	Length of memory protection field (4 bits)

17. For BLISS, the functions of the symbols in the previous three items are combined into a single name used to reference an arbitrary datum. Names are of the following form, where *x* is the same as *t* for standard-sized data (B, W, L, and Q) and *x* stands for V for arbitrary and bit fields:

    structure$*x*_*field-name*

    The macro includes the offset, position, size, and sign extension suitable for use in a BLISS field selector. Most typically, this name is defined by the following BLISS statement:

```
    MACRO
structure$V_field-name=
        structure$t_field-name,
        structure$V_field-name,          !VAX MACRO V
                                         ! bit field definition
        structure$S_field-name,
        <sign extension> %;
```

18. Public structure mask names are of the form

    structure$M_*field-name*

The value of the public symbol is a mask with bits set for each bit in the field. This mask is not right-justified. Rather, it has *structure*$V_*field-name* zero bits on the right. The following examples show this form of symbol name:

PCB$M_RES	Mask of process residency field
VA$M_BYTES_PER_PAGELET	Mask of bytes within pagelet field
PTE$M_PROT	Mask of memory protection field

19. Public structure constant names are of the form

    structure$K_*constant-name*

    The following examples show this form of symbol name:

PCB$K_LENGTH	Length (in bytes) of software process control block
EIHD$K_EXE	Image is executable (in image header)

    For historical reasons, many of the constants used by the executive have the letter C instead of K to indicate that the object data type is a constant. Examples of this form of symbol name are

DYN$C_PCB	Structure type is software process control block
EXE$C_SYSEFN	Common system event flag
PRT$C_URKW	Protection code of user read, kernel write

20. PSECT names are of the form

    facility$*mnemonic*

    When these names are put into a library, they have the form

    _*facility*$*mnemonic*

    The following examples show symbols of the form *facility*$*mnemonic*:

COPY$COPY_FILE	File copying main routine program section
DCL$ZCODE	Program section that contains most code for the Digital command language interpreter
DBG$CODE	Program section containing debugger routines

    This convention is not adhered to as strictly as the other naming conventions because PSECT names control the way that the linker allocates virtual address space. Names are often chosen to affect the relative locations of routines and the data they reference.

    Some sample PSECT names from the Run-Time Library show examples of the form _*facility*$*mnemonic*:

_LIB$CODE	General library (read-only) code section
_STR$DATA	Data section in the string manipulation library
_OTS$CODE	Code portion of language-independent support library

D.2 Object Data Types

Table D–1 shows some of the letters used to indicate data types or reserved for various other purposes. N, P, and T strings are typically variable-length. In structures or I/O records, they frequently contain a byte-sized digit or character count preceding the string. If so, the location or offset is to the count. Counted strings cannot be passed in procedure calls. Instead, a string descriptor must be generated.

Table D–1 Letters and the Data Types They Indicate

Letter	Data Type or Use
A	Address
B	Byte integer
C	Character[1]
D	Double precision floating
E	Reserved to Digital
F	Single precision floating
G	G_floating-point values
H	H_floating-point values
I	Reserved for integer extensions
J	Reserved to customers for escape to other codes
K	Constant
L	Longword integer
M	Field mask
N	Numeric string (all byte forms)
O	Octaword
P	Packed string
Q	Quadword integer
R	Reserved for records (structure)
S	Field size
T	Text (character) string
U	Smallest unit of addressable storage
V	Field position (VAX MACRO) Field reference (BLISS)
W	Word integer
X	Context-dependent (generic)
Y	Context-dependent (generic)
Z	Unspecified or nonstandard

[1] In many of the symbols used by VMS, C is used as a synonym for K. Although K is the preferred indicator for constants, many constants used by VMS are indicated by a C in their name. Some constants, such as lengths of data structures, have both a C form and a K form.

D.3 Facility Prefix Table

Table D–2 lists some of the facility prefixes used by Digital-supplied software. This list is not inclusive and is intended to show examples of several facility prefixes. Each facility name has a unique facility code.

Note that bit <27>, the customer facility bit, is clear in all the facility codes listed here. Customers are free to use any of the facility codes listed here, provided that they set bit <27>. The default action of the message compiler is to set this bit.

The location of the facility code within a status code and the meaning of the other fields in the status code are described in the *VMS Utility Routines Manual*.

Table D–2 Facility Names and Their Prefixes

Prefix	Description	Condition <27:16>
	Executive and System Processes	
SS	System service status codes	0
CLI	Command language interpreters	3
JBC	Job controller	4
OPC	Operator communication	5
ERF	Error logger format process	8
	Run-Time Library Components	
SMG	Screen management routines	18
LIB	General-Purpose Library	21
MTH	Mathematics Library	22
OTS	Language-independent object time system	23
FOR	FORTRAN Run-Time Library	24
SORT	SORT procedures	28
STR	String manipulation procedures	36
	Utilities and Compilers	
DBG	Symbolic debugger	2
LIN	VMS Linker	100
DIF	File Differences Utility	108
PAT	Image File Patch Utility	109

Individual products such as compilers also have unique facility codes formed from the product name.

Structure name prefixes are typically local to a facility. Refer to the individual facility documentation for its structure name prefixes. Individual facility structure names do not cause problems, because these names are not global and are therefore not known to the linker. They become known at assembly or compile time only by explicit invocation of the macro defining the facility structure.

For example, the macro $FORDEF defines all the status codes that can be returned from the FORTRAN support library. The facility code of 24 is included in the upper 16 bits of each of the status codes defined with this macro.

Digital provides a registration service for customer facility names. For information on this service, contact

Product Registrar—GSF1-1/C19
Digital Equipment Corporation
5 Wentworth Drive
Hudson, NH 03079-4298

D.4 Further Information

- *The DIGITAL Guide to Software Development*, Digital Press, Order Number EY-C178E-DP.

Fold here and tape

PURCHASE A PRELIMINARY VOLUME NOW AND SAVE ON THE FINAL EDITION LATER

☐ Yes, I recently purchased a preliminary volume of ***VMS for Alpha Platforms Internals and Data Structures*** and would like to receive information about a special discount on the final edition. In the final book, all chapters will be updated to describe one particular released version of VMS for Alpha platforms.

Please send details of this discount to:

Name _____

Company _____

Address _____

_____

_____

Phone ( ) _____

☐ Yes, I would also like to receive the latest Digital Press catalog.

(Do not tear) Fold here

Please tape all three edges.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 29 BURLINGTON, MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL PRESS
DIGITAL EQUIPMENT CORPORATION
ONE BURLINGTON WOODS DRIVE OFO
BURLINGTON, MA 01803-9572

Please tape all three edges.

This volume is the first publication to explain VMS operating system support for Digital Equipment Corporation's recently announced 64-bit RISC architecture, known internally by the code name Alpha. Written for computer professionals who need this detailed knowledge as early as it can be made available, the Preliminary Edition is based on *VAX/VMS Internals and Data Structures: Version 5.2* (Digital Press, 1991). Volume 1 contains twelve chapters and an appendix. Subsequent volumes will be published as more chapters are completed.

Topics covered in this volume include:

I/O Architecture ■ Interrupts and Exceptions ■ Hardware Interrupts and Machine Checks ■ Software Interrupts ■ System Service Dispatching ■ ASTs ■ Synchronization Techniques ■ Overview of the I/O Subsystem ■ I/O System Services ■ I/O Processing ■ The Modular Executive ■ Bootstrap Processing ■ Naming Conventions

Ruth E. Goldenberg is a member of the VMS Operating System Development Group at Digital Equipment Corporation. She is principal author of both the Version 4 and Version 5 editions of *VAX/VMS Internals and Data Structures.*

Saro Saravanan is also a member of Digital's VMS Operating System Development Group. He was a contributor to *VAX/VMS Internals and Data Structures: Version 5.2.*

digital™

Digital Press
One Burlington Woods Drive
Burlington, Massachusetts 01803

ORDER NUMBER EY-L466E-P1
DP ISBN 1-55558-095-5
PH ISBN 0-13-948480-9